VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

VYLEPŠENÍ REKONSTRUKCE DATOVÝCH TYPŮ PŘI ZPĚTNÉM PŘEKLADU

DATA TYPE RECONSTRUCTION IMPROVEMENTS IN RETDEC DECOMPILER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ADAM VENGER**
AUTHOR

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

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2018/2019

Zadání bakalářské práce

Student:	**Venger Adam**
Program:	Informační technologie
Název:	**Vylepšení rekonstrukce datových typů při zpětném překladu** **Data Type Reconstruction Improvements in RetDec Decompiler**
Kategorie:	Překladače

Zadání:

1. Studujte problematiku reverzního inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s principy tzv. manglování jmen (name mangling) při překladu. Studujte schémata manglování používaná nejrozšířenějšími překladači populárních programovacích jazyků. Studujte možnosti tzv. demanglování takto generovaných jmen.
3. Seznamte se se zpětným překladačem RetDec společnosti Avast a s technikami rekonstrukce datových typů v něm používaných.
4. Navrhněte nástroj provádějící proces opačný k manglování (demanglér) pro vybranou množinu programovacích jazyků a jejich překladačů. Navrhněte rozšíření technik rekonstrukce datových typů v nástroji RetDec, která využijí takto získané informace.
5. Po konzultaci s vedoucím a konzultantem implementujte nástroje a rozšíření navržená v předchozím bodě.
6. Vytvořené řešení důkladně otestujte sadou testů, včetně reálných programů, nebo vzorků potenciálně škodlivých programů.
7. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Dokumentace k vybraným překladačům.
- Dokumentace k projektům RetDec, LLVM atd.
- Dle doporučení vedoucího či konzultanta.

Pro udělení zápočtu za první semestr je požadováno:

- První čtyři body zadání a rozpracování pátého bodu.

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

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

Abstrakt

Nový škodlivý software sa objavuje neustále. Pre jeho efektívnu analýzu a boj proti nemu sú potrebné nástroje ako dekompilátor. Dekompilácia je zložitý problém a jej zlepšenie vyžaduje využitie všetkých dostupných informácií obsiahnutých v binárnom súbore. Určité programovacie jazyky vyžadujú pre správnu kompiláciu zdrojových kódov zakódovanie mien použitých symbolov. Do mena funkcií sú pridané napríklad typy parametrov a konvencia volania. Tento proces sa nazýva mangling. Táto práca sa zaoberá procesom opačným tzv. demanglingom a jeho využitím pre zlepšenie nástroja pre spätný preklad strojového kódu do vyššieho jazyka, RetDec. Vytvorená knižnica umožňuje demangling symbolov vytvorených populárnymi kompilátormi jazyka C++ a jazyka Delphi. Kombinuje vlastné riešenie s existujúcim v podobe demangleru projektu LLVM. Existujúca knižnica demangleru nástroja RetDec bola nahradená novou, ktorej výsledky sú spoľahlivejšie. Na základe informácií získaných z kódovaných symbolov bola rozšírená rekonštrukcia dátových typov, čo spôsobilo spresnenie výstupu dekompilácie.

Abstract

New malware is being continuosly developed. For its effective analysis and fight against it, tools such as decompilers are needed. Decompilation, however, is difficult problem to solve. For the improvement of its results, all of the information contained in binary files needs to be used. Some programming languages require encoding of some symbols in order to be compiled correctly. For example, when compiling functions, the parameter data types and the calling convention are encoded to the function name. This process is called mangling. Thesis deals with reverse process called demangling and its utilization for improvement of the RetDec decompiler. Created library allows demangling of symbols created by popular C++ and Delphi compilers. It combines custom solution with an existing one in the form of LLVM project demangler. Existing demangler library in RetDec was replaced with the new one, results of which are much more reliable. The reconstruction of data types was expanded to use the information obtained from encoded symbols, which resulted in more accurate decompilation.

Kľúčové slová

reverzné inžnierstvo, RetDec, demanglovanie, demangler, rekonštrukcia dátových typov

Keywords

reverse engineering, RetDec, demangling, demangler, mangling, name decoration, data type reconstruction

Citácia

VENGER, Adam. *Vylepšení rekonstrukce datových typů při zpětném překladu.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dušan Kolář, Doc. Dr. Ing.

Vylepšení rekonstrukce datových typů při zpětném překladu

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Doc. Dr. Ing. Dušana Koláře. Ďalšie informácie mi poskytol Ing. Peter Matula. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

....
Adam Venger
15. mája 2019

Poďakovanie

Ďakujem vedúcemu mojej práce doc. Dr. Ing. Dúšanovi Kolářovi za jeho odborné vedenie počas jej písania. Takisto by som chcel poďakovať môjmu konzultantovi zo spoločnosti Avast Ing. Petrovi Matulovi za jeho odbornú pomoc a poskytnuté informácie.

Obsah

1 Úvod **3**

2 Reverzné inžinierstvo **4**
2.1 Použitie reverzného inžinierstva 4
2.2 Princípy reverzného inžinierstva 5
2.3 Nástroje používané pri reverznom inžinierstve 5
2.3.1 Nástroje pre dynamickú analýzu 5
2.3.2 Nástroje pre statickú analýzu 6
2.3.3 Populárne nástroje pre analýzu kódu 7

3 Manglovanie mien **9**
3.1 Aplikačné binárne rozhranie 10
3.2 Výskyt manglovaných symbolov 10
3.3 Skúmané jazyky 11
3.3.1 Jazyk C++ 11
3.3.2 Delphi 12
3.4 Manglovacie schémy 13
3.4.1 Itanium 13
3.4.2 Microsoft Visual C++ Compiler 14
3.4.3 Borland 15

4 RetDec **18**
4.1 Štruktúra 18
4.1.1 Predspracovanie 18
4.1.2 Jadro 19
4.1.3 Backend 20
4.2 Demangling v nástroji RetDec 20
4.2.1 Aktuálna implementácia 20
4.3 Rekonštrukcia dátových typov 21
4.3.1 Knižnica ctypes 21
4.3.2 Analýza parametrov 21

5 Návrh novej knižnice pre demangling **22**
5.1 Požiadavky na nový demangler 22
5.2 Existujúce riešenia 22
5.2.1 Testovanie 23
5.3 Borland demangler 24
5.4 Rekonštrukcia dátových typov 24

5.4.1 Rozšírenie analýzy parametrov funkcií 25
5.5 Návrh rozhrania knižnice demangleru 27
5.5.1 Rozhranie 27
5.5.2 Prístup k demangleru 27
5.5.3 LLVM demangler 28

6 Implementácia knižnice demangleru a rozšírení analýzy parametrov 29
6.1 Demangler 29
6.1.1 LLVM Demangler 29
6.1.2 Borland demangler 30
6.2 Rekonštrukcia dátových typov 32
6.2.1 Konverzia syntaktického stromu do ctypes 32
6.2.2 Konverzia ctypes do LLVM IR 32
6.2.3 Rozšírenie analýzy 34

7 Testovanie 35
7.1 Demangling 35
7.1.1 Porovnanie úspešnosti s pôvodnou knižnicou demangleru 35
7.1.2 Porovnanie rýchlosti s pôvodnou knižnicou demangleru 36
7.2 Využitie typových informácií pri dekompilácií 36

8 Záver 39

Literatúra 40

A Príloha 41
A.1 Gramatika manglovacej schémy prekladačov Embacadero. 41

B Obsah přiloženého pameťového média 43

Kapitola 1

Úvod

Analýza škodlivého softwaru je zložitý problém. Pri jeho skúmaní je najčastejšie k dispozícií len binárny súbor. Porozumieť mu, však môže byť problém aj pre skúsených odborníkov. Jazyky vyššej úrovne sú, na druhej strane, jednoducho pochopiteľné aj pre menej skúsených. Spätná kompilácia podozrivých súborov do vyššieho jazyka sa preto javí ako ideálne riešenie. Dekompilátor RetDec je jeden z nástrojov úspešne používaný pre tento účel. Výsledky jeho dekompilácie, ale nie sú dokonalé. Proces spätného prekladu je náročný a pre správne výsledky musí použiť všetky dostupné zdroje informácií. Jednou z oblastí, v ktorej prekladač nevyužíva všetky informácie, sú mená symbolov.

Pri preklade určitých jazykov sa pre mená symbolov používa tzv. mangling, ktorý do mena symbolu zakóduje rôzne cenné informácie ako počet a typ parametrov. Je potrebný primárne pri preklade jazykov umožňujúcich preťažovanie funkcií (function overloading). Umožňuje rozlišovanie jednotlivých preťažovaných funkcií na úrovni binárneho kódu. Dekódovanie takýchto mien sa nazýva demangling. Výsledkom demanglingu býva bežne deklarácia pôvodného symbolu v textovej podobe. Demangling môže byť využitý pri analýze binárneho kódu ako nápoveda pri skúmaní napríklad funkcií. Tieto informácie je však možné použiť aj počas dekompilácie pre skvalitnenie výsledkov. Program prevádzajúci demangling musí zvládať spracovať symboly obsahujúce aj jednoduché, bežne využívané jazykové konštrukcie a dátové typy a aj tie zložitejšie. Prípadné nesprávne dekódovanie môže pri použití týchto informácií negatívne ovplyvniť dekompiláciu.

Aktuálna knižnica demangleru implementovaná v spätnom prekladači produkuje nesprávne výsledky a v niektorých prípadoch dokonca spôsobuje pád aplikácie. Nesprávne výsledky sú produkované už pri demanglingu relatívne jednoduchých jazykových konštrukcií. RetDec, v prípade úspešného demanglingu symbolu, tento dekódovaný symbol pridáva ako komentár do výsledného výstupu. Pre zlepšenie výsledkov dekompilácie, kde je každá informácia dôležitá, je potrebné tento zdroj využiť lepšie. Informácie o dátových typoch parametrov je možné použiť pre presnejšiu rekonštrukciu funkcií.

Táto práca stručne približuje svet reverzného inžinierstva, používaných nástrojov a spätný prekladač RetDec. Rozoberá pojmy ako dekorácia mien alebo mangling spolu s ich implementáciami v často používaných prekladačoch. Na príkladoch sú vysvetlené rozdiely v manglingu pri použití týchto prekladačov. Vysvetľuje kde a kedy môžu byť manglované symboly nájdené. Stredobodom pozornosti práce je proces demanglingu a problémy s ním spojené. Práca dokumentuje implementáciu novej knižnice vykonávajúcej demangling. Ďalej sú popísané možnosti využitia týchto informácií pre zlepšenie dekompilovaného výstupu a ich finálna implementácia. Na záver je nová knižnica demangleru porovnaná s tou pôvodnou, zlepšenia výsledkov dekompilácie sú overené na reálnych programoch.

Kapitola 2

Reverzné inžinierstvo

Chikofsky a Cross vo svojej práci *Reverse engineering and design recovery: a taxonomy* [6] definoval reverzné inžinierstvo ako:

> Reverzné inžinierstvo je proces, pri ktorom sa skúma vnútorná štruktúra a fungovanie istého systému. Vytvára sa reprezentácia vo vyššej forme abstrakcie.

Účel tohto skúmania môže byť snaha o zlepšenie dizajnu, o jeho zreprodukovanie, alebo snaha o odhalenie skrytých prvkov, ktoré nie sú na prvý pohľad zrejmé. Reverzné inžinierstvo je len proces skúmania a získavania poznatkov. Skúmaný systém sa nemodifikuje. Využíva sa pri absencii nejakého poznatku. Má široké pole pôsobnosti od mechanického inžinierstva, cez chemické, až po softwarové. Táto práca sa sústredí na softwarové reverzné inžinierstvo.

Kapitola približuje reverzné inžinierstvo, použitie a nástroje používané pri ňom. Popis vychádza z kníh *Reversing: Secrets of Reverse Engineering* [8] a *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* [12].

2.1 Použitie reverzného inžinierstva

Pri vývoji softwaru môže nastať viac prípadov, kedy sa bez reverzného inžinierstva nezaobídeme. Pri využívaní proprietárneho softwaru tretích strán, alebo práci s proprietárnym operačným systémom sa neraz vývojár dostane do situácie, kedy je funkcia nedostatočne zdokumentovaná alebo nie je zdokumentovaná vôbec. Vtedy je jediná možnosť použiť reverzné inžinierstvo na zistenie detailov rozhrania.

V iných odvetviach je reverzného inžinierstvo najčastejšie použité pri vývoji konkurenčného produktu. Softwarové produkty bývajú ale veľmi komplexné, preto väčšinou nedáva ekonomický zmysel reverzovať celý produkt a na kopírovanie konkurenčných produktov sa používa len zriedka. Často býva ekonomickejšie vyvinúť produkt od základu alebo ho licencovať, ako ho reverzovať celý. Výnimkou sú len unikátne dizajny a algoritmy, ktoré by bolo príliš drahé navrhnúť.

Reverzné inžinierstvo je možné využiť aj pri vyhodnocovaní kvality programov. Kvalitu kódu a programu najlepšie vyhodnotíme kontrolou zdrojového kódu. To ale pri proprietárnych produktoch nie je možné. Reverzovanie nám samozrejme nepovie toľko, ako by odhalila kontrola zdrojového kódu. Ale reverzné inžinierstvo použité spolu s dôkladným testovaním je najlepší spôsob ako preveriť kvalitu produktu.

Techniky reverzného inžinierstva sú často používané pri skúmaní a vyhodnocovaní bezpečnosti enkryptovacích programov. Takto je možné odhaliť prípadnú chybu, ktorá by znížila náročnosť dekrypcie dát pre ich potenciálne zneužitie.

Reverzovanie je vo veľkej miere využívané pri vývoji malwaru a pri boji proti nemu. Vývojári malwaru ho využívajú pri hľadaní slabín v operačných systémoch a inom software. Nájdenie takého slabého miesta môže viesť k infekcií malwarom, ktorý môže mať za úlohu získanie prístupu k citlivým informáciám, alebo dokonca získanie plného prístupu k systému. Tomuto sa snažia zabrániť vývojári antivírusového softwaru. Využívajú reverzovanie na porozumenie vzorkám malwaru. Tieto informácie potom využijú pri zhodnotení škody, rozsahu nakazenia, možností jeho odstránenia a obrany pred ďalším nakazením.

2.2 Princípy reverzného inžinierstva

Pri skúmaní systémov môžeme postupovať dvomi spôsobmi. Systémy je možné skúmať dynamicky, pozorovaním systému pri práci a skúmaním jeho interakcií s okolím. Alebo staticky, teda analýzou dát a vnútornej štruktúry.

Dynamická analýza je analýza na systémovej úrovni. Pomocou dynamickej analýzy je možné zistiť všeobecnú štruktúru programu a nájsť užšie miesta záujmu. Väčšinu informácií o interakcii programu s okolím získavame z operačného systému pod ktorým je program spustený. On zabezpečuje ovládanie hardwaru, komunikáciu medzi programami a prístup k užívateľským dátam. Pri moderných systémoch nie je bežne možné získať prístup k týmto prostriedkom bez vedomia operačného systému. Všetky interakcie preto musí umožňiť on. Dynamická analýza môže ale znamenať aj sledovanie okolia systému. Je možné sledovať jeho sieťovú komunikáciu pomocou zachytávania paketov (packet sniffing) alebo komunikáciu hardwarových komponentov odpočúvaním zberníc.

Statická analýza je analýza na úrovni kódu. Pri nej sú dôkladne skúmané jednotlivé inštrukcie a ich náväznosť. Používa sa na dôkladné preskúmanie prvkov a ich prípadnú rekonštrukciu do vyššej formy abstrakcie. Programy nie sú spúšťané čo zamedzuje ohrozeniu potenciálne škodlivým softwarom a nie je potrebná príprava izolovaného prostredia. Schopnosti takejto analýzy sú však obmedzené. Rôzne metódy obfuskácie sťažujú zisťovanie toho, čo v konečnom dôsledku program bude vykonávať po jeho nahraní do pamäte.

2.3 Nástroje používané pri reverznom inžinierstve

V reverzovaní pomáhajú rôzne nástroje, ktorých cieľ je abstrahovať a zhromažďovať dostupné informácie. Tieto nástroje však len prezentujú určité informácie. Analytik z nich musí vedieť vyvodiť znalosti o skúmanom systéme.

2.3.1 Nástroje pre dynamickú analýzu

Dynamická analýza malwaru vyžaduje spustenie skúmaného súboru. Pred tým než je spustený, je potrebná príprava vhodného prostredia, ktoré odstráni riziká spojené so spúšťaním škodlivého softwaru. K tomuto účelu sa používajú rôzne virtualizačné a sandboxové prostredia.

Medzi nástroje pre dynamickú analýzu patrí napríklad debugger. Debugger umožňuje zastaviť vykonávanie programu pri splnení istej podmienky, najčastejšie dosiahnutie dopredu určenej inštrukcie. Umožňuje prečítanie hodnôt uložených v registroch a premenných. Jedny z najpopulárnejších debuggerov sú GDB a Microsoft Visual Studio Debugger

pre Windows. Jeho primárny účel je vývoj softwaru, ale uplatnenie nájde aj pri reverznom inžinierstve.

Dôležitým aspektom malwaru je komunikácia s okolím. Takto môže byť odhalený spôsob akým sa šíri alebo spôsob akým dokáže zneužiť nakazený systém. Pre tento účel je vhodné použiť nástroje na monitorovanie sieťovej komunikácie. Analýzou zasielaných paketov je možné zistiť kam sú zasielané získané dáta, čo je obsahom týchto správ alebo slabé miesto využívané pre jeho ďalšie šírenie. Najpopulárnejší z týchto nástrojov je Wireshark.

2.3.2 Nástroje pre statickú analýzu

V minulosti jediný spôsob vývoja softwaru bolo písanie v jazyku symbolických inštrukcií „Assembly language". Je to pomenovanie pre skupinu jazykov. Jednotlivé príkazy, sú textová reprezentácia inštrukcií procesora. Pre každú architektúru sa líši a nie je vzájomne kompatibilný. Moderný vývoj využíva jazyky vyššej úrovne, ktoré sú následne prekladané do assembleru a do binárneho kódu, alebo do bytekódu, ktorý je spustený na virtuálnom stroji. Tieto súbory sú generované súborom nástrojov ako sú prekladač a linker. Pre analýzu binárneho kódu je používaný disassembler, ktorý binárny súbor preloží naspäť do jazyka symbolických inštrukcií. Kód generovaný týmto spôsobom nebýva intuitívny pre človeka, pretože prekladač často používa optimalizačné techniky, ktoré môžu výrazne zhoršiť čitateľnosť kódu a nie sú na prvý pohľad zrejmé. Reverzný inžinier musí vedieť pracovať s takýmto kódom, pretože je to často jediná možnosť.

Disassemblery

Disassemblovanie je väčšinou prvý krok pri statickej analýze. Aj tento zdanlivo priamočiary krok môže spôsobovať problémy. Existujú techniky na sťaženie procesu disaassemblovania. Hlavne pri architektúrach s premenlivou dĺžkou inštrukcií. Jednou z nich je vloženie dát ako náhodného šumu na nedosiahnuteľné miesto v kóde. Toto spôsobí problémy disassemblingu prevádzanému lineárne. Lineárny disassembling začína analýzu od určitého miesta lineárne a všetky dáta na ktoré narazí sú intrepretované ako inštrukcie. Po dosiahnutí miesta s vloženým šumom by všetky ďalšie inštrukcie boli preložené nesprávne. Preto je inteligentnejší spôsob prechádzať inštrukcie rekurzívne, ale aj tu je možné naraziť na podobné techniky. Disassembler je súčasťou interaktívnych debuggerov.

Dekompilátory

Dekompilátor alebo spätný prekladač je nástroj pre preklad binárnych súborov do vyššieho programovacieho jazyka. Fungujú opačne ako kompilátory. Často sa používa v spolupráci s disassemblerom. Dekompilácia dokáže výrazne uľahčiť pochopenie zložitých, alebo dokonca obfuskovaných konštrukcií v assemblery. Použitie dekompilátoru vie pomôcť aj pri analýze assembleru, ktorý analytik nepozná. Dekompilátor môže byť skvelý nástroj, ale tento spätný preklad nikdy nebude dokonalý. Pri kompilácií sa veľa informácií stratí a niektoré informácie nie je možné jednoznačne rekonštruovať. Výsledky závisia ako od použitého pôvodného vysokoúrovňového jazyka a konkrétnych programov, tak aj od formátu preloženého kódu. Napríklad .NET programy kompilované do MSIL majú typicky veľmi dobré výsledky dekompilácie. Binárny kód pre x86 toľko informácií neobsahuje, preto býva jeho dekompilácia menej úspešná. Výsledky závisia aj od použitia respektíve nepoužitia obfuskácie.

Problémy dekompilácie

Jeden z problémov, ktoré dekompilácia nikdy nebude vedieť vyriešiť je rekonštrukcia mien parametrov a premenných. Často sa kompiláciou strácajú aj mená funkcií. Tento problém bude ďalej rozoberaný v časti 3.2, kde je upresnený výskyt týchto symbolov. Aj napriek problémom dokáže dekompilácia natívneho binárneho kódu výrazne pomôcť pri reverzovaní a urýchliť túto prácu.

2.3.3 Populárne nástroje pre analýzu kódu

Radare2

Radare2[1] je súbor nástrojov pre reverzné inžinierstvo. Pôvodne začal ako nástroj pre forenznú analýzu. Obsahuje nástroje pre statickú aj dynamickú analýzu ako debugger, disassembler, nástroje pre extrakciu informácií ako sú sekcie binárneho súboru a mnoho ďalších. Veľká výhoda tohto frameworku je pokročilá možnosť skriptovania a automatizácie. Framework obsahuje aj nástroj pre dekompiláciu.

IDA

Interaktívny disassembler a debugger IDA[2], je populárny nástroj, ktorý môže prácu výrazne uľahčiť. Umožňuje zobrazovať graf riadenia toku a ďalšie grafy. Ponúka možnosť skriptovania a rozšírenia pomocou pluginov. Medzi komunitou reverzných inžinierov patrí k najpopulárnejším nástrojom pre statickú analýzu malwaru. IDA Pro je platený nástroj vyvíjaný spoločnosťou Hex-Rays. Staršia verzia IDA je dostupná aj zdarma, ale obsahuje obmedzenia. Umožňuje použitie nástroja len pre skúmanie obmedzenej množiny architektúr. Spoločnosť Hex-Rays neposkytuje technickú podporu a produkt nie je možné využívať pre komerčné účely. Platená verzia nástroja IDA Pro obsahuje plugin dekompilátoru s názvom Hex Rays.

Hex Rays Decompiler

Hex Rays[3] je proprietárny dekompilátor, ktorý je súčasťou nástroju IDA Pro. Výsledky jeho dekompilácie sú jedny z najlepších. Funguje v spolupráci s IDA, teda je možné pre zlepšenie výsledkov dekompilácie využiť informácie dodané IDA používateľom. To dokáže ďalej spresniť podobu výstupného kódu.

Ghidra

Ghidra[4] je pre verejnosť nový framework pre analýzu potenciálne škodlivého kódu, vyvinutý Americkou NSA. Jej kód bol zverejnený 4. Apríla 2019. Framework obsahuje pokročilé nástroje pre disassembling, dekompiláciu, tvorbu grafov a skriptovanie. Ponúka možnosť interaktívneho prostredia pre užívateľa, ale aj mód automatizovaný. Za krátky čas, ktorý je nástroj dostupný pre verejnosť, sa ukazuje ako veľmi schopný nástroj pre reverese engineering schopný konkurovať IDA Pro.

[1] https://rada.re/r/
[2] https://www.hex-rays.com/products/ida/index.shtml
[3] https://www.hex-rays.com/products/decompiler/index.shtml
[4] https://github.com/NationalSecurityAgency/ghidra

RetDec

Retargetable Decompiler alebo RetDec[5] je open-source dekompilátor vyvíjaný spoločnosťou Avast. Je zložený z viacerých nástrojov, ktoré je možné použiť samostatne. Existuje možnosť ho využiť ako plugin pre IDA Pro podobne ako Hex Rays Decompiler. Podrobne je popísaný v kapitole 4.

[5]https://github.com/avast/retdec

Kapitola 3

Manglovanie mien

Pojem mangling vznikol zo slovného spojenia „to mangle“, teda prekrútiť, alebo zmrzačiť. *Proces transformácie mien zdrojového kódu do mien objektového súboru sa nazýva name mangling.* [11] Tieto mená sa inak nazývajú aj dekorované. Ďalej bude práca tento pojem referovať ako manglovanie. Kapitola vychádza z manuáluálov rôznych prekladačov a ABI, z článku Agnera Foga *Calling conventions for different C++ compilers and operating systems* [9] a knihy *Linkers and loaders* [11]. Tieto zdroje sú kombinované, porovnané a overené vlastnými experimentmi.

Zmysel manglingu je primárne zamedzenie menných konfliktov funkcií, tried a iných objektov pri preklade programov. Tento proces je najviac viditeľný pri menách funkcií. Funkcia je blok inštrukcií uložených na určitej adrese v pamäti. Preložený program teda nepracuje priamo s funkciami, ale s adresami inštrukcií na ktoré sa skáče pomocou inštrukcií ako call a jump. Pred procesom linkovania tieto adresy nemusia byť známe, pretože sa môžu nachádzať v inom prekladanom module. Jednotlivé mená funkcií a adresy na ktorých sa nachádzajú inštrukcie funkcie sú uvedené v tabuľke symbolov. Tieto mená musia byť unikátne, aby sa program na symboly mohol jednoznačne odkazovať.

Proces manglingu a tvar výsledného mena je závislí na prekladanom jazyku. Je vykonávaný tak, aby funkcie, ktoré môžu mať rozličnú definíciu, boli jednoznačne rozlíšiteľné podľa tohto manglovaného mena. Preto do manglovaného mena je typicky zakódovaný menný priestor, názov funkcie a dátové typy parametrov. Ďalší efekt manglingu je, že umožňuje pri linkovaní jednoducho vykonať kontrolu, či boli všetky objekty v moduloch deklarované rovnako.

Mangling je dôležitý pre preklad jazykov ktoré využívajú preťažovanie funkcií (function overloading) ako Java, C++, Delphi. Pri preťažovaní funkcií môže mať viac funkcií rovnaké meno, ale líšia sa v počte alebo dátovom type parametrov. Tieto funkcie môžu mať rôzne definície. Preto je dôležité na úrovni assembleru a tabuľky symbolov takéto funkcie rozlišovať. Pri preklade sa všetky dátové typy parametrov zakódujú do dekorovaného mena. Všeobecne platí že nie je možné preťažovať funkcie len na základe návratovej hodnoty funkcie. Preto väčšina manglovacích schém túto informáciu pri tvorbe mena nepoužíva. Existujú ale výnimky, ktoré budú rozoberané v sekcií 3.3.

3.1 Aplikačné binárne rozhranie

Aplikačné binárne rozhranie alebo ABI špecifikuje rozhranie medzi užívateľským kódom, systémom, knižnicami a rôznymi modulmi. Definuje aj spôsob volania funkcií medzi rôznymi modulmi, spracovania výnimiek a iné konvencie. To zahňuje spôsob uloženia vstavaných a užívateľsky definovaných dátových objektov v pamäti a aj kompilátorom generované objekty ako virtuálne tabuľky. ABI je definované na úrovni binárneho kódu. Takisto môže zahrňovať aj spôsob manglovania symbolov.

3.2 Výskyt manglovaných symbolov

V tejto sekcií bude vysvetlený proces kompilácie so zameraním na mangling, ako sa používajú manglované mená a kde ich je možné nájsť. Príklady budú ukázané na jazyku C++ preloženom prekladačom GCC [5] pod operačným systémom Linux. Ukážky budú demonštrovať transformáciu mien funkcií. Je to z dôvodu ich vyššej komplexnosti v porovnaní s manglovaním mien tried. Pri použití iných prekladačov a iných operačných systémov je postup obdobný. Takisto je obdobný pri iných jazykoch kompilovaných do binárnych spustiteľných súborov formátov PE a ELF.

Proces prekladu prebieha v dvoch fázach:

1. **Kompilácia do objektových súborov**

   Pri kompilácií je najprv každý modul kompilovaný zvlášť do objektového súboru. Tabuľka symbolov objektových súborov obsahuje všetky externé symboly. V jazyku C++ sú to všetky, ktoré nie sú explicitne označené ako statické. V tejto fáze ešte nie je možné presne povedať adresu funkcií definovaných v iných moduloch. Preto sa v tabuľke relokácií nachádza položka odkazujúca sa na manglované meno a miesto kam sa má adresa symbolu pri linkovaní vložiť.

2. **Linkovanie**

   Linkovanie môže prebiehať buď staticky, kedy budú všetky použité moduly a knižnice súčasťou jedného spustiteľného súboru. Alebo knižnice môžu byť linkované dynamicky, až pri spustení.

   Pri statickom linkovaní aj napriek tomu že všetky adresy budú známe už pri preklade, prekladač GCC necháva v tabuľke symbolov všetky symboly. Tieto symboly sa ale môžu explicitne zmazať a funkčnosť programu to neovplyvní. Vtedy už spustiteľný súbor nebude obsahovať žiadne manglované mená. Toto správanie nastáva len pri prekladoch do formátu ELF a pri použití prekladačov GCC a Clang.

   Iný prípad ale nastáva pri preklade dynamicky linkovaných knižníc. Tabuľky symbolov týchto knižníc a takisto programy, ktoré volajú funkcie týchto knižníc musia obsahovať manglované mená funkcií. Mená funkcií nemôžu byť z tabuľky odstránené. Dynamický linker potrebuje podľa mena získať adresu funkcií až pri spustení.

Práca skúma manglované mená a ich podobu na kóde, vzniknutom kompiláciou do assembleru. Manglovanými symbolmi sú označené napríklad bloky inštrukcií reprezentujúce funkciu. Tento postup bol zvolený pre jednoduchú dostupnosť a získavanie manglovaných mien. Tieto mená sa od mien vyskytujúcich sa vo finálnom spustiteľnom súbore alebo dynamicky linkovanej knižnici nelíšia, preto použitie tohto postupu neovplyvní výsledky

práce. Cieľová architekúra rovnako výsledok neovplyvňuje a bolo dostatočné analyzovať assembler jednej architektúry. Preto bol postup analýzy assembleru považovaný za valídny. Kód preložený do assembleru ukazuje tabuľka 3.1.

```
int foo(int a, char b, float c);

int main(int argc, char **argv) {
   return foo(1, 'b', 1.0);
}
```

```
⋮
movl $98, %esi
movl $1, %edi
call _Z3fooicf@PLT
nop
⋮
```

Obr. 3.1: Porovnanie zdrojového kódu (vľavo) a časť assembleru produkovaného nástrojom GCC vzniknutá jeho kompiláciou (vpravo). Dôležitá je časť `_Z3fooicf`, čo je manglovaná podoba funkcie `int foo(int a, char b, float c)`.

3.3 Skúmané jazyky

Práca sa zameriava na vylepšenia dekompilátoru RetDec. Nástroj je určený pre dekompiláciu binárnych súborov. Z jazykov využívajúcich mangling sa pre tvorbu malwaru najčastejšie používajú jayky C, C++ a Delphi. Manglované symboly jazyka C sú veľmi jednoduché a neobsahujú žiadne typové informácie, ktoré by mohli pomôcť pri rekonštrukcií. Ako bude ďalej ukázané, symboly C++ a Delphi majú veľa spoločné a obsahujú množstvo informácií, ktoré je možné ďalej pri dekompilácií použiť. Preto sa práca bude ďalej venovať len manglovaným symbolom produkovaným prekladačmi jazyka C++ a Delphi.

Každý jazyk má vlastný súbor pravidiel, podľa ktorých sa vykonáva mangling. Tento súbor pravidiel sa nazýva manglovacia schéma. Pri jej návrhu sa musel brať ohľad na potreby konkrétneho jazyka. Manglovacie schémy sa líšia aj v rámci jedného jazyka. Schéma teda takisto závisí na použitom jazyku ako aj na použitom prekladači. Príkladom na rozdiely v rámci jedného jazyka sú schémy prekladačov GCC a Microsoft Visual C++ Compiler. Súbory produkované týmito prekladačmi sú vzájomne nekompatibilné z viacerých dôvodov. Nekompatibilita manglovacích schém indikuje že aj ostatné aspekty ABI budú nekompatibilné.

3.3.1 Jazyk C++

C++ je kompilovaný, silne typovaný, multiparadigmový jazyk vyvíjaný od osemdesiatych rokov. Vznikol ako rozšírenie jazyka C. Preto s ním zdieľa veľkú časť syntaxe a vstavané dátové typy. Podporuje procedurálne programovanie, objektovo orientované programovanie a generické programovanie. Jazyk obsahuje výnimky (exceptions) a virtuálne funkcie. Tieto vlastnosti robia z C++ veľmi rozsiahly jazyk. Navyše je pod aktívnym vývojom a zväčšuje sa každou ďalšou verziou. Aktuálna verzia v čase písania práce je C++17 (ISO-/IEC 14882:2017 [1]) a má 1605 strán. Všetkych tieto aspekty jazyka môžu mať dopad na podobu manglovaných mien.

Zo špecifikácie jazyka je možné vyčleniť niekoľko oblastí záujmu, ktoré je potrebné preskúmať a môžu mať dopad na výslednú podobu dekorovaného mena:

- Mená a menné priestory – Jazyk umožňuje vytvárať menné priestory. Pri manglovaní sa musia použiť plne kvalifikované mená. Plne kvalifikované meno obsahuje postupne všetky menné priestory, v ktorých je symbol deklarovaný.
- Vstavané jednoduché dátové typy – Jednoduché dátové typy sú často používané, preto bývajú jednoducho zakódované, väčšinou pomocou jedného znaku.
- Užívateľom definované triedy (dátové typy).
- Konvencie volania – Prekladače ponúkajú možnosť použiť špecifickú konvenciu volania, tá je sa väčšinou objavuje aj v manglovanom mene.
- Šablóny – Do mena šablóny musia byť zakódované argumenty s ktorými bola vytvorená. To zahrňuje dátové typy, ukazatele, referencie, dokonca číselné výrazy. V prípade použitia šablón vo funkciách býva explicitne uvedený jej návratový typ a to aj pri použití schémy, ktorá ho bežne do manglovaného mena nevkladá.
- Metódy tried a ich odlíšenie od funkcií – Ako bude ďalej v kapitole ukázané, niektoré schémy tieto objekty rozlišujú.

Pre analýzu boli vybrané najpopulárnejšie prekladače jazyka C++: GCC, Clang, Microsoft Visual Compiler a Embarcadero C++ Compiler. V prípade open-source prekladačov je relatívne jednoduché zistiť manglovaciu schému, ale dva zo skúmaných prekladačov boli proprietárne. Ich skúmanie je časovo náročnejšie a vyžaduje podrobnú znalosť jazyka.

3.3.2 Delphi

Delphi je vysokoúrovňový, kompilovaný a silne typovaný jazyk. Podporuje objektovo-orientovaný návrh a šablónové programovanie. Pôvodne vyvíjaný spoločnosťou Borland, aktuálne pod spoločnosťou Embarcadero.

Veľa jeho vlastností je podobných s jazykom C++, čo bolo využité pri použitej manglovacej schéme. Prekladače jazyka C++ od Embarcadero a jazyka Delphi používajú rovnakú manglovaciu schému. Podrobne je rozoberaná v sekcii 3.4.3. Predmetom skúmania prekladača jazyka Delphi boli hlavne rozdiely medzi jazykmi ako napríklad manglovanie vstavaných typov jazyku Delphi, ktoré sa v C++ nenachádzajú.

Jazyk Delphi obsahuje typy, ktoré nemajú svoj ekvivalent v štandartnom C++. Tieto typy sú ale definované v knižniciach prekaldača Embarcadero C++ Compiler. V dokumentácií jazyka Delphi [4] sú všetky tieto typy vysvetlené. Väčšinou sú implementované ako šablóny.

Delphi	C++
Currency	System::Currency
Comp	System::Comp
ShortString	System::SmallString<255>
AnsiString	System::AnsiStringT<0>
UnicodeString	System::UnicodeString
WideString	System::WideString
RawByteString	System::AnsiStringT<65535>
UTF8String	System::AnsiStringT<65001>

Tabuľka 3.1: Vstavané dátové typy jazyka Delphi, ktorých ekvivalenty neexistujú v štandardnej knižnici C++ (vľavo) a ich C++ ekvivalent z knižníc prekladača Embarcadero C++ Compiler.

3.4 Manglovacie schémy

Rôzne prekladače pristupujú rôzne k stratégií manglovania symbolov. Niektoré produkujú mená, ktoré obsahujú čo najviac informácií, niektoré naopak kódujú len minimum informácií potrebných k rozlíšeniu symbolov.

Deklarácia	GCC	Microsoft Visual Compiler
`int foo(int a, int b)`	`_Z3fooii`	`?foo@@YAHHH@Z`

Tabuľka 3.2: Príklad nekompatibilných schém.

V tabuľke 3.2 je na jednoduchom príklade ukázaný rozdiel v symboloch produkovaných prekladačmi GCC a Microsoft Visual Compiler. Na prvý pohľad je možné vidieť, že symbol z prekladaču Microsoft obsahuje viac neznámych symbolov, čo by mohlo naznačovať že bude obsahovať viac informácií. Jednotlivé schémy sú podrobnejšie popísané v sekciách 3.4.1, 3.4.2 a 3.4.3.

Rôzne prekladače môžu ale používať aj rovnakú schému. Príklad na kompatibilné prekladače sú prekladače jazyka C++: GCC a Clang ako vidieť v tabuľke 3.3.

Deklarácia	GCC	Clang
`int foo(double d)`	`_Z3food`	`_Z3food`

Tabuľka 3.3: Príklad kompatibilných schém.

Oba prekladače sú kompatibilné s Itanium C++ ABI. Pri nekompatibilite ABI bývajú používané aj nekompatibilné manglovacie schémy. Použitie rovnakej schémy pri jej nekompatibilite môže spôsobiť úspešné linkovanie, ale neočakávané správanie pri spustení takto vytvoreného programu.

3.4.1 Itanium

Itanium C++ ABI je súčasť ABI pôvodne vytvoreného pre platformu Itanium. Bolo však navrhnuté aby ho bolo možné využívať na širokom spektre platforiem. ABI je popísané vo voľne dostupnom manuále [7]. Popis ABI obsahuje formálnu definíciu gramatiky vysvetľujúcej syntax a sémantiku manglovaných mien. Toto ABI dodržujú dva veľmi rozšírené

open-source prekladače: GCC [5] a Clang [3]. Používa ho aj MinGW vývojové prostredie pre Windows, ktoré obsahuje port GCC pre tento operačný systém. Pod týmto názvom bude práca ďalej referovať manglovciu schému pre kompilátory kompatibilné s Itanium C++ ABI.

`void foo(void)`	`_Z3foov`
`void bar::foo(int i)`	`_ZN3bar3fooEi`
`void Baz::foo(double d1, double d2)`	`_ZN3Baz3fooEdd`
`template <class T> double f (T a, T b)`	`_Z1fIiEdT_S0_`

Tabuľka 3.4: Príklad manglovania v schéme Itanium.

V tabuľke 3.4 sú uvedené jednoduché príklady ako vyzerajú manglované mená pomocou tejto schémy. Z nich je možné vyvodiť základné znaky ako rozlíšiť Itanium symboly od symbolov produkovaných inými schémami. Základné znaky:

- Manglovaný symbol začína na `"_Z"`.
- Mená sú predchádzané údajom o ich dĺžke.
- Klasifikátory mien sú ohraničené znamkmi `"N"` a `"E"`.

symbol	význam
`_Z`	značí začiatok manglovaného symbolu
`N`	hovorí že bude nasledovať kvalifikované meno
`3`	označuje dĺžku mena
`Bar`	je menný priestor/kvalifikátor nasledujúceho symbolu
`3`	je znova dĺžka mena
`foo`	je nekvalifikované meno
`E`	označuje koniec kvalifikovaného mena
`j`	je typ prvého parametru funkcie – unsigned integer
`d`	je typ druhého parametru funkcie – double

Tabuľka 3.5: Vysvetlenie častí jednoduchého manglovaného symbolu v schéme Itanium na manglovanom symbole `_ZN3Bar3fooEjd`.

Na príklade v tabuľke 3.5 sú podrobne rozobrané rôzne aspekty schémy Itanium. Porovnaním pôvodného mena `int Baz::foo(int i, double d)` a manglovaného mena môžeme vidieť že niektoré informácie boli stratené. Boli to konkrétne návratový typ funkcie a mená parametrov. Kvalifikátory mena a meno sú v rovnakom poradí ako v pôvodnej deklarácií. V niektorých implementáciách môžu manglované mená začať prefixom `__Z`. Je to v prípade kompilácie pre operačný systém Mac OS X.

3.4.2 Microsoft Visual C++ Compiler

Dokumentácia k manglovacej schéme prekadaču Microsoft Visual C++ Compiler nie je voľne dostupná. Prekladač je proprietárny a preto sa pri skúmaní manglovacej schémy tohto prekladaču používajú techniky reverzného inžnierstva. Aj napriek tomu je táto schéma relatívne dobre zmapovaná. Jej najlepšia dostupná dokumentácia sa nachádza v už spomínanom článku *Calling conventions* [9, str. 29].

`void foo(void)`	`?foo@@YAXXZ`
`void bar::foo(int i)`	`?foo@bar@@YAXH@Z`
`void Baz::foo(double d1, double d2)`	`?foo@Baz@@QAEXNN@Z`
`template <class T> double f (T a, T b)`	`?$f@H@@YANHH@Z`

Tabuľka 3.6: Príklad manglovania v schéme Microsoft Visual Compiler.

Základné znaky:

- Symboly začínajú "?" alebo "??".
- Meno a jednotlivé menné priestory sú oddelené znakom "@".
- Meno a menné priestory sú v opačnom poradí v porovnaní s C++ zdrojovým kódom.
- Menná časť je ukončená dvojicou znakov "@@".
- Do manglovaných mien funkcií je zakódovaný aj návratový typ funkcie.

3.4.3 Borland

Embarcadero C++ Compiler je ďalší z proprietárnych prekladačov, s vlastnou manglovacou schémou. Pôvodne prekladač C++ firmy Borland je aktuálne vyvíjaný spoločnosťou Embarcadero pod názvom Embarcadero C++ Compiler. Prekladače firmy Embarcadero jazyku C++ a prekladač jazyku Dephi rovnakej firmy majú veľa spoločné. Majú spoločné ABI, prekladačom Delphi je možné generovať knižnice s hlavičkovými súbormi C++ a C++ kód je možné vložiť priamo do zdrojových kódov Delphi. S tým súvisí že používajú rovnakú manglovaciu schému. Ďalej spoločnú manglovaciu schému pre tieto 2 prekladače budem volať historickým, zaužívaným menom Borland.

Manglovacia schéma Borland bola pri písaní práce analyzovaná najpodrobnejšie. Boli analyzované symboly produkované prekladačom C++ a Delphi. Pri porovnaní sa nepodarilo nájsť žiadny rozdiel v ich podobe. Vždy bolo možné nájsť ekvivalentné deklarácie v oboch jazykoch, ktoré produkujú rovnaké manglované mená. Po tomto zistení bola ďalej schéma skúmaná pri preklade jazyka C++. Skúmanie C++ umožňuje lepšie preskúmať schému manglovania. C++ je komplexnejší jazyk a pre testovanie je možné vytvoriť testovaciu sadu spolu s testovacou sadou pre iné schémy.

Manglovacia schéma bole skúmaná experimentálne. Nie je dostatočne dobre popísaná v iných zdrojoch, preto bola najprv budovaná všeobecná predstava o podobe symbolov. Skúmala sa podoba vstavaných dátových typov, definovaných, ukazateľov a referencií. Výstupy boli porovnávané pri použití rôznych konvencií volania. Neskôr boli testované šablóny a ich podoba. Pre zaručenie systematického postupu bola použitá referencia jazyka C++ na stránkach výrobcu kompilátora [2].

`void foo(void)`	`@foo$qv`
`void bar::foo(int i)`	`@bar@foo$qi`
`void Baz::foo(double d1, double d2)`	`@Baz@foo$qdd`
`template <class T> double f (T a, T b)`	`@%f$i%$qii$d`

Tabuľka 3.7: Príklad manglovania v schéme Borland.

Základné znaky:

- Dekorované mená začínajú znakom "@".
- Jednotlivé mená a menné priestory sú oddelené znakom "@".
- Menná časť je zakončená znakom "$".
- Návratový typ je kódovaný len v prípade použitia šablónových funkcií.
- Mená a menné priestory sú v rovnakom poradí ako v C++ zdrojovom súbore.

symbol	význam
@	značí začiatok manglovaného symbolu
Baz	kvalifikátor nasledujúceho mena
@	meno ešte pokračuje
%	začiatok šablóny
foo	meno šablóny
$	koniec mena a začaitok argumentov
i	argument šablóny – integer
%	koniec šablóny
$	koniec mena, začiatok funkčného typu
qqr	konvencia volania – fastcall
3	nasleduje pomenovaný typ, ktorého manglované meno má dĺžku 3 znaky
Baz	dátový typ – trieda Baz
$	koniec parametrov, nasleduje návratový typ funkcie
i	dátový typ – integer

Tabuľka 3.8: Vysvetlenie jednotlivých častí jednoduchého manglovaného symbolu v schéme Borland na manglovanom symbole `Baz%foo$i%$qqr3Baz$i`.

Na príklade v tabuľke 3.8 je možné vidieť, že kvalifikátory mena a meno sú v rovnakom poradí ako v pôvodnej deklarácií. Pomenované typy sú predchádzané údajom o dĺžke ich manglovaného mena. Pri manglovaní šablónovej funkcie sa nestratí údaj o návratovom type funkcie, inak áno. Šablóny sú ohraničené znakom %.

Chyby prekladača súvisiace s manglovaním

Pri analýze Borland schémy boli objavené chyby prekladaču spôsobené nesprávnym návrhom schémy manglovania. Schéma manglovania by mala fungovať tak, že dva rôzne vstupy nikdy nespôsobia identické výstupy (zanedbávané sú mená parametrov). Takéto prípady však nastali. Prekladač v jednom prípade túto chybu detekoval a nedovolil preklad. V druhom prípade bol program preložený, ale jeho správanie bolo v rozpore so štandardom jazyka C++.

- Kľúčové slovo restricted – Prvý prípad bol pri použití kľúčového slova restricted. Pri vytvorení dvoch symbolov, ktoré by spôsobili konflikt sa objaví chyba a prekladač odmietne takto vytvorený zdrojový kód preložiť. Podľa štandardu jazyka C++ bol tento kód platný a zdrojové kódy boli preložiteľné pomocou iných prekladačov. Toto spôsobilo problém pri demanglingu. Keďže kľúčové slovo restricted je menej používané ako referencie, tak riešenie bolo demangling symbolu `r` ako referencie. Jediný prípad

kedy je možné s určitosťou povedať že sa jedná o použitie kľúčového slova restricted resp. referencie je pri použití restricted referencie. Výsledok manglovania je v tomto prípade `rr`. Dôvod je že podľa štandardu C++ nie je povolená referencia na referenciu.

- Užívateľom definovaný literál – Druhý prípad nastal pri použití *user defined literal* operátora. Jeho volanie je vysvetlené na nasledujúcom príklade: `25_abc`. Pri tomto použití je volaná funkcia `int operator _abc (int)`. Operátor je manglovaný ako `@li_abc$qi`. Je možné vytvoriť funkciu, ktorá vyprodukuje manglované meno spôsobujúce konflikt. Tá vyzerá nasledovne: `int li_abc(int)`. V tomto prípade ale na konflikt prekladač neupozorní. Preloží len definíciu funkcie, ktorá sa nachádzala v zdrojovom kóde skôr a tú použije pri volaní oboch funkcií. Toto je v rozpore so štandardom jazyka C++. Chyba vzniká unikátnym spôsobom manglovania tohto operátora, ktoré sa líši od menglovania iných operátorov. Táto chyba má potenciálne vážne následky. Prejaví sa až pri testovaní/používaní programu.

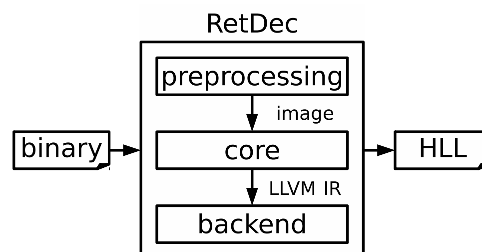
Kapitola 4

RetDec

Jedným z nástrojov používaných pri reverznom inžinierstve je RetDec [10]. RetDec (Retargetable Decompiler) je dekompilátor strojového kódu do vysokoúrovňového jazyka. Začal ako proprietárny software pod vývojom AVG v spolupráci s Fakultou Informačných Technológií VUT. V roku 2017 boli zdrojové kódy zverejnené pod MIT licenciou. Je aktívne vyvíjaný a používaný spoločnosťou Avast. Podporuje 32 a 64 bitové architektúry x86, ARM, PowerPC a MIPS.

4.1 Štruktúra

RetDec tvorí sada nástrojov, ktoré je možné použiť samostatne.



Obr. 4.1: Prehľad častí nástroju RetDec.

Dekompilátor pracuje v troch fázach:

1. Predspracovanie
2. Jadro
3. Backend

4.1.1 Predspracovanie

Predspracovanie zabezpečuje prevod vstupného binárneho súboru do jednotnej internej reprezentácie. RetDec vie spracovať rôzne formáty vstupných súborov: ELF, PE, COFF, Mach-O, Intel HEX, AR a surové dáta. Po prevode do internej reprezentácie nasleduje analýza pomocou YARA pravidiel, ktorá detekuje použitý kompilátor a potenciálne použitie kompresie. Tieto informácie sú použité na vytvorenie konfiguračného súboru používaného v ďalších fázach dekompilácie. Následne, je interná reprezentácia transformovaná do obrazu

pamäte pomocou simulácie loaderu. Je to z dôvodu aby dekompilátor pracoval s dátami tak ako by pracoval operačný systém. Keď bolo detekované použitie packeru, tak je upackovaný pomocou príslušnej knižnice. Je možné spracovať aj debugovacie informácie vo formátoch DWARF a PDB.

4.1.2 Jadro

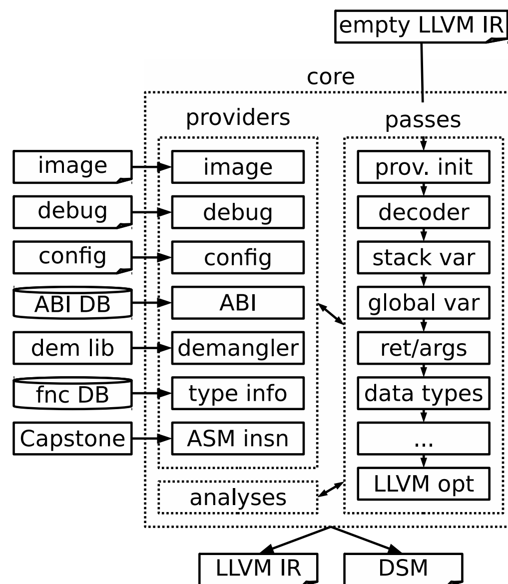
Úlohou jadra je transformácia načítaného obrazu pamäti vytvoreného v prvom kroku do LLVM IR. Jadro dekompilátoru tvorí nástroj kombinujúci prechody LLVM a svoje vlastné.

LLVM je kompilačný framework, ktorý funguje spôsobom že vysokoúrovňový jazyk najprv preloží do LLVM IR, čo je jazyk podobný assembleru alebo java bytekódu. Následne nad týmto kódom vykonáva analýzy a optimalizácie vo viacerých prechodoch. Každý prechod slúži na istú optimalizáciu. Tento spôsob je veľmi modulárny. RetDec používa len niektoré prechody LLVM. Dôvodom je, že úlohou LLVM prechodov je znižovanie abstrakcie smerom k binárnemu kódu. RetDecu má presne opačnú úlohu – zvyšovanie abstrakcie smerom k vysokoúrovňovému jazyku.

Moduly jadra sa delia do 3 skupín:

- *Providers (Poskytovatele)* slúžia ako rozhranie k externým súborom (obraz prekladaného súboru, konfigurácia, debugovacie informácie), knižniciam (demangler, disasssembler) a databázam (ABI modely).
- *Passes (prechody)* sa používajú na transformácie a optimalizácie. Každý prechod analyzuje vstup, upraví ho a jeho výstup je predaný ďalšej časti.
- *Analyses (Analyzátory)* sú moduly, ktoré analyzujú LLVM objekty, ale nemodifikujú ich. Tieto informácie sú dostupné pre ďalšie použitie LLVM prechodom.

Jadro najprv vytvorí prázdny modul, do ktorého sa budú ukladať dekompilované inštrukcie LLVM IR. Po naplnení sa bude kód postupne modifikovať pomocou prechodov ako napríklad prechod na rekonštrukciu globálnych premenných, premenných na zásobníku a prechod na rekonštrukciu triednej hierarchie.



Obr. 4.2: Prehľad jadra dekompilátoru.

4.1.3 Backend

Jeho úlohou je konverzia LLVM IR do vysokoúrovňového jazyka. Pri tejto konverzii je využívaný medzikrok, kedy sa LLVM IR preloží do Backend IR (BIR). BIR vytvára štruktúry podobné abstraktnému stromu. Kód je štruktúrovaný do konštrukcií vyšších programovacích jazykov (if-then-else). Nad kódom je spoustený súbor optimalizácií. Následne je z BIR generovaný výstup v jazykoch C alebo Python. Je možné generovať aj graf volaní, graf toku riadenia a rôzne štatistiky.

4.2 Demangling v nástroji RetDec

Demangling je proces opačný k manglingu. Demanling a jeho ďalšie použitie na základe tejto práce budú podrobne popísané v kapitole 5. RetDec obsahuje knižnicu vykonávajúcu demangling. Prístup k nej zabezpečuje demangler provider. Je možné ho použiť aj ako jednoduchý program v terminály.

V aktuálnej podobe sa využíva na 3 miestach:

- Pri menách funkcií získaných z tabuľky symbolov.
- Pri menách funkcií získaných z debug informácií.
- Pri menách tried a rekonštrukcií triednej hierarchie.

Výsledkom tohto demanglingu je vždy komentár pri funkcií, ktorej meno bolo možné demanglovať alebo pri menách tried, tiež ako komentár.

```
// Address range: 0x11b6 - 0x11cd
// Demangled: foo(int, char)
int32_t _Z3fooic(int32_t a1, int32_t a2) {
    // 0x11b6
    int32_t v1;
    __x86_get_pc_thunk_ax(g2, v1, a1);
    return a2 + a1;
}
```

```
// Address range: 0x11b6 - 0x11cd
int32_t _Z3fooic(int32_t a1, int32_t a2) {
    // 0x11b6
    int32_t v1;
    __x86_get_pc_thunk_ax(g2, v1, a1);
    return a2 + a1;
}
```

Obr. 4.3: Porovnanie dekompilácie pri použití manglingu (vľavo) a bez neho (vpravo).

V ukážke 4.3 je možné vidieť použitie demanglingu a jeho vplyv na generovaný výstup. Pri jeho použití pribudol komentár s demanglovaným menom pri definícií funkcie. Iný vplyv demangling na výstup nemá. Z demanglovaného mena je možné vidieť že pôvodná funkcia mala dva parametre typov integer a char. Analýza RetDecu ale tieto parametre detekovala inak a informácie z demanglingu zostali nevyužité.

4.2.1 Aktuálna implementácia

Aktuálna implementácia bola navrhovaná ako rozšíriteľná, preto sú manglovacie schémy uložené v textových súboroch. To sa však ukázalo ako nepraktické riešenie, ktoré nesplnilo tento účel rozšíriteľnosti. Parser gramatík z textového súboru je ťažko udržiavateľný a implementačné chyby sú ťažko opraviteľné. Aktuálna implementácia generuje chybné výstupy, ktorých príklady sú uvedené v tabuľke 4.1. Má problémy s demanglingom templatov.

Neobsahuje všetky demanglovacie pravidlá pre danú množinu schém a niektoré vstupy dokonca spôsobujú pád aplikácie. Z týchto dôvodov bolo rozhodnuté nahradiť celú knižnicu demangleru.

Dekorované meno	Očakávaný výstup	Výstup demangleru
`?f@@YAXPB_WZZ`	`void __cdecl f(wchar_t const *,...)`	`void __cdecl f(wchar_t const *)`
`_ZL1fh`	`f(unsigned char)`	–
`_ZTI5cName`	`typeinfo for cName`	–
`_ZdlPvS_`	`operator delete(void*, void*)`	–

Tabuľka 4.1: Príklad chybných výstupov.

4.3 Rekonštrukcia dátových typov

Rekonštrukcia dátových typov je zložitá úloha. Musia sa skúmať rôzne miesta ako registre a zásobník, kde môžu byť uložené dáta. Na základe spôsobu prístupu k týmto pamäťovým miestam je možné odhadnúť dátový typ uložnej informácie. Jedným z najdôležitejších miest, kde je potrebná informácia o dátových typoch je rozhranie funckií a to konkrétne dátových typov parametrov funkcie. RetDec je pod aktívnym vývojom a momentálne jeho schopnosť rekonštrukcie kódu vytvoreného jazykom C++ je obmedzená. Problémom sú komplexné štruktúry a triedy jazyka.

4.3.1 Knižnica ctypes

RetDec pre vnútornú reprezentáciu dátových typov a funkcií používa knižnicu `ctypes`. Štruktúry knižnice `ctypes` sú následne konvertované do jazyka LLVM IR. Knižnica podporuje reprezentáciu pomerne jednoduchých dátových typov, ktoré sú používané v jazyku C. Sú to celočíselné dátové typy, číselný typ s pohyblivou desatinnou čiarkou, ukazateľ, funkčný dátový typ a funkcie. Nepodporuje reprezentáciu pokročilých dátových typov ako sú L-value, R-value referencie alebo triedy. Táto reprezentácia sa používa pri načítaní informácií o funkciách a dátových typoch zo súboru. Súbor je vytvorený pri predspracovaní. Reprezentácia sa využíva len ako medzikrok. Nevyužíva sa na žiadne analýzy.

4.3.2 Analýza parametrov

V súčasnosti je počet a dátový typ parametrov analyzovaný na základe volaní funkcií. Všetky volania sú zhromaždené, skúmajú sa registre a potenciálne argumenty predávané na zásobníku. Na základe prieniku množín zozbieraných zmenených informácií pred každým volaním a architektúry sa odhaduje použitá konvencia volania a aj počet a typ parametrov. Nad touto množinou informácií prebieha analýza a filtrácia.

Kapitola 5

Návrh novej knižnice pre demangling

Demangling je proces opačný k manglingu. Z mena, ktoré sa nachádza v spustiteľných súboroch, knižnicach, objektových súboroch a ktoré je pre človeka takmer nečitateľné, dostávame meno používané v zdrojových súboroch. Demangling je závislý na použitej manglovacej schéme. Každá schéma musí mať demangler vytvorený priamo pre onu konkrétnu schému, inak by hrozil nesprávny preklad. Demanglované meno by malo mať rovnakú podobu ako pôvodná deklarácia. Keďže nie všetky pôvodné informácie zo zdrojového súboru sa zachovávajú pri preklade, nie vždy bude možné pomocou demanglingu z manglovaného mena vytvoriť kompletnú originálnu deklaráciu. V manglovanej forme funkcie sa nevyskytujú napríklad informácie o menách parametrov a bežne sa tu nevyskytuje návratový typ.

5.1 Požiadavky na nový demangler

Pre navrhovaný demangler sa stanovili nasledujúce požiadavky:

- Pôvodný demangler implementuje manglovacie schémy pre prekladače s Itanium ABI, Microsoft Visual C++ Compiler a Borland Compiler. Nový demangler bude musieť vedieť demanglovať minimálne tieto 3 demanglovacie schémy.
- Musí mať jednotné rozhranie.
- Musí byť spoľahlivý, ľahko udržiavateľný.
- Mal by byť jednoducho rozšíriteľný.
- Nemôže spôsobiť pád aplikácie.
- Výsledky demanglingu by mali byť použiteľné aj pre ďalšiu analýzu a zlepšenie výsledkov dekompilácie aj nad rámec dekódovania deklarácie mien funkcií.

5.2 Existujúce riešenia

Pre prekladače využívajúce manglovaciu schému Itanium a pre prekladače Microsoft Visual C++ Compiler existuje už viacero knižníc, ktoré s rôznou úspešnosťou vykonávajú demangling mien. Práca analyzuje rôzne existujúce implementácie a ich možnosti implementácie

do nástroja RetDec. Pre schému prekladaču Borland neexistuje vhodná knižnica, preto musí byť implementovaná nová.

5.2.1 Testovanie

Pre skúmanie bolo vybraných viacero knižníc. Niektoré z nich uvádzajú kompatibilitu so schémami Itanium aj Microsoft.
Skúmané knižnice pre Itanium boli: *LLVM demangler, GCC dcp-demangle, demumble.*
Pre schému Microsoft VS: *LLVM demangler, pdbparse, demumble, msvcfilt.*

Pri ich testovaní sa zohladňovali nasledovné vlastnosti:

- Použité knižnice musia byť aktívne vyvíjané, alebo overené početnou užívateľskou základňou.
- Korektnosť výstupov.
- Stabilitu knižnice.
- Možnosť rozšírenia knižnice pre použitie v RetDecu.
- Kompatibilita licencie s licenciou používanou RetDecom.

mscfilt a pdbparse

msvcfilt a *pdbparse* sú už dlho nepodporované a nikdy nemali veľkú užívateľskú základňu. Ich implementácia neumožňovala jednoduché rozšírenie pre potreby RetDecu. To ich vylúčilo z množiny možných kandidátov.

demumble

Knižnica *demumble* mala taktiež malú užívateľkú základňu, ale bola aktívne vyvíjaná. Implementuje demangling pre Itanium aj Microsoft schému. Avšak pri testovaní bol nájdený súbor korektných vstupov, ktoré spôsobili pád knižnice. Ich opravenie by vyžadovalo netriviálne zásahy do aplikácie.

GCC dcp-demangle

Jedna z dvoch knižníc, ktoré sa dostali do užšieho výberu. Knižnica demangleru od GCC je často používaná. Je aktívne vyvíjaná a má širokú základňu užívateľov. Je súčasťou GNU binary utilities pod názvom c++filt. Veľmi dobre otestovaná a podporuje demangling symbolov schémy Itanium a jazyka Java.

LLVM demangler

Knižnica LLVM demangleru obsahuje demangling symbolov Itanium aj Microsoft. Počas testovania knižnice LLVM demangleru sa objavili vstupy, ktoré spôsobili nesprávne chovanie. Pri demanglovaní mien Itanium schémy boli nájdené vstupy spôsobujúce nesprávny výstup. Tento výstup bol spôsobený pomerne zložitou syntaxou jazyka C++ pri deklarovaní ukazatela na funkciu. Chyba sa však prejavila len nesprávnym poradím výstupných symbolov, vnútorná reprezentácia bola správna. Ďalšia chyba nastala pri pokuse o demangling nevalídneho vstupu demanglerom pre Microsoft. V týchto prípadoch knižnica dorazila

do neočakávaného stavu a havarovala. Kontrola bola vykonávaná nesprávnym spôsobom. Tieto problémy boli ale jednoducho riešiteľné a vďaka aktívnemu vývoju veľkej komunity je pravdepodobné že sa výsledky budú len zlepšovať. Výraznou výhodou tejto knižnice bola jednoduchá možnosť rozšírenia pre ďalšie použitie v dekompilátore.

Výsledky

Záverom skúmania bolo, že demangler projektu LLVM je vhodným kandidátom na implementáciu do dekompilátoru. Dôvodov je viacero:

- Jeho výsledky boli podľa testov dostačujúce a všetky známe nedostatky boli priamočiaro riešiteľné.
- Je testovaný a vylepšovaný open-source komunitou.
- Jeho spôsob implementácie umožňuje pomerne jednoduché rozšírenia potrebné pre ďalšiu analýzu dátových typov získavaných z dekorovaných mien.
- Obsahuje implementáciu Itanium aj Microsoft VS demanglingu.
- Zdrojové kódy sú voľne dostupné pod kompatibilnou open-source licenciou Apache.
- Je súčasťou veľkého, aktívne vyvíjaného a podporovaného projektu LLVM.
- RetDec už využíva staršiu verziu LLVM v jadre analýzy, teda po aktualizovaní knižnice nemusia pribúdať ďalšie závislosti na knižnice tretích strán a problémy s licenciou.

5.3 Borland demangler

Pôvodná knižnica demangleru používaná v RetDecu podporuje demangling schémy Borland. Inú knižnicu, ktorá by túto úlohu zvládala lepšie sa nepodarilo nájsť. Preto je potrebné tento nástroj implementovať od začiatku.

Pre manglovaciu schému bola vytvorená gramatika (príloha A.1). Na jej základe prebieha analýza manglovaných mien. Bude využívaná syntaktická analýza zhora nadol (top-down parsing) a upravená metóda rekurzívneho zostupu. Pri analýze bude v niektorých prípadoch kontrolovaných viac znakov naraz. Oproti kontrole po jednom znaku tento postup prinesie určité spomalenie, no výrazne zjednoduší implementáciu a zvýši prehľadosť a udržateľnosť kódu. To boli jedny z hlavných požiadaviek pre novú knižnicu. Počas syntaktickej analýzy bude súčasne vykonávaná aj sémantická analýza. Výsledky demanglingu budú ďalej používané pri rekonštrukcií dátových typov, čo musí byť zohľadnené v návrhu. Knižnica bude preto pre vnútornú reprezentáciu demanglovaného symbolu používať štruktúru syntaktického stromu.

5.4 Rekonštrukcia dátových typov

Aktuálna analýza dátových typov parametrov funckií v RetDecu nie je dokonalá. Ako je rozoberané v sekcií 4.3, zdroje poskytujúce informácie o dátových typoch pre dekompiláciu sú obmedzené. Manglované mená môžu slúžiť ako ďalší zdroj informácií. Obsahujú typové informácie o parametroch, ktoré by mohli tejto analýze pomôcť a spresniť ju.

Rekonštrukcia typov pomocou demanglovania symbolov môže výrazne pomôcť pri dekompilácií binárnych súborov vytvorených prekladačmi GCC a Clang. Tie v tabuľke symbolov zachovávajú aj položky, ktorých prítomnosť v tabuľke pre správny beh programu nie je potrebná. Na prítomnosť týchto informácií sa však nedá spoliehať, pretože je možné ich odstrániť. Odstránenie týchto symbolov je predpokladané hlavne pri škodlivom software.

5.4.1 Rozšírenie analýzy parametrov funkcií

Konvencia volaní

Prvý krok k zlepšeniu prekladu pomocou demanglingu symbolov môže byť dodanie informácie o použitej konvencií volania funkcie. To obmedzí rozsah prehľadávaných registrov a hodnôt na zásobníku a zamedzí nutnosti použitia heuristík k odhadnutiu konvencie. Použitá konvencia volaní sa pri schéme manglovania Itanium na výsledku nikdy neprejaví, teda sa nedá použiť pri zlepšení analýzy. Pri prekladačoch Borland a Microsoft je však možné s istotou určiť konvenciu volania. Táto informácia bude dodaná dekompilátoru pred začatím analýzy parametrov.

Typy parametrov získané demanglovaním

Pri manglovaní je zakódovaný dátový typ parametrov a ich počet do jedného mena. Dekompilátor pre vnútornú reprezentáciu používa knižnicu ctypes rozoberanú v sekcií 4.3.1. Táto knižnica je vhodný spôsob ako reprezentovať funkcie a ich parametre počas ďalšej analýzy. Je ale navrhovaná pre typy jazyka C a pre jazyk C++ ju bude potrebné rozšíriť. Bude rozšírená o typy L-value referencia, R-value referencia a pomenovaný typ.

Problémy pri detekcií parametrov

Jazyk C++ má ale okrem explicitných parametrov aj parametre implicitné. Ich výskyt z manglovaného mena nie je možné detekovať. Dôvodom je, že jazyk C++ umožňuje použitie objektovo-orientovaného modelu. Spôsob implementácie modelu je, že ako prvý parameter býva predaný ukazateľ na objekt. Tento parameter je implicitný a preto sa v manglovanom mene nenachádza. Zo samotného manglovaného mena teda nie je možné zistiť či je daná funkcia metódou triedy a obsahuje implicitný argument alebo nie je. Porovnanie názvu menného priestoru funckie so známymi triedami na túto detekciu nestačí. Dôvodom je možnosť vytvárania statických funkcií triedy, ktoré tento parameter neobsahujú.

To znamená že je potrebné nájsť spôsob ako rozlíšiť metódy od bežných funkcií. Pri správnej detekcií počtu parametrov dekompilátorom RetDec by bolo možné tento úkaz detekovať. Preto aj napriek úspešnému demanglingu a získaní informácií o dátových typoch parametrov je potrebné vykonať analýzu parametrov. Potom bude potrebné tieto informácie porovnať.

Podmienky pre určenie či je funkcia zároveň metódou triedy sú nasledovné:

- Oproti parametrom detekovaním demanglingu by mala analýza parametrov nájsť o jeden parameter naviac.
- Na prvom mieste by mal byť ukazteľ na integer o veľkosti podľa architektúry (32 bitová architektúra – int32). Dôvodom prečo by to bol integer je ten, že RetDec namiesto rekonštrukcie zložitých dátových typov použije predvolený dátový typ – integer.
- Meno funkcie by muselo obsahovať meno triedy ako kvalifikátor.

Keď sú tieto podmienky splnené, mohli by sme povedať, že funkcia je triedna metóda. Ako prvý parameter by bol použitý ukazateľ na integer s veľkosťou podľa architektúry, pomenovaný `this` a ďalšie typy parametrov by boli použité na základe demanglingu. V prípade že by RetDec podporoval triedy ako dátové typy, by ako dátový typ mohol byť použitý ukazateľ na triedu. Pri skúmaní tejto možnosti sa však ukázalo, že riešenie nebude validne. Jazyk C++ a takisto aj množstvo ďalších využíva *return value opimization* (RVO). Tento pojem označuje optimalizáciu, používanú keď funkcia vracia štruktúru alebo dátový typ, ktorý sa nezmestí na miesto, kde by štandardne bola uložená návratová hodnota funkcie. Preto je miesto na uloženie návratovej hodnoty alokované volajúcim funkcie a funkcii je predaný ukazateľ na toto miesto ako parameter. Tento parameter sa takisto nenachádza v manglovanom mene a z neho tento úkaz nie je možné detekovať. Parameter je pridaný na začiatok a z pohľadu dekompilátoru bude mať rovnakú podobu ako by mal ukazateľ na objekt *this*. Preto sú tieto dva prípady na úrovni analýzy vstupov nerozlíšiteľné.

Tento problém je riešený tak, že v prípade, kedy je parametrov detekovaných o 1 viac ako hovorí manglovaný symbol, tak prvý typ zostane rovnaký a ďalšie sa doplnia podla demanglingu. Meno parametru sa nemení. V prípade kedy je detekovaných parametrov o 2 viac ako hovorí manglovaný symbol, vtedy môžeme kontrolovať vlastnosti prvého parametru podľa pôvodného plánu. Vtedy je možné ako prvý parameter vložiť ukazateľ *this*, ako druhý parameter ukazateľ *result* a následne parametre podľa demanglovaného mena. Ak by analýza našla ešte viac parametrov ako o dva viac, je vysoká pravdepodobnosť, že analýza bola nesprávna.

V budúcnosti by bolo možné spresniť analýzu použitia RVO. Malo by byť možné určiť prípady kedy sa s určitosťou táto optimalizácia nepoužije alebo naopak použije. V prípade správnej analýzy by bolo možné presnejšie určiť typ implicitného parametru, prípadne ho aj premenovať na *this* alebo *result*. Toto správanie je ovlyvňované veľkým počtom faktorov:

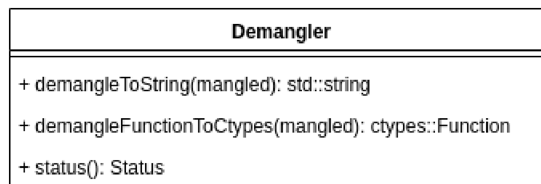
- Konvencia volania ovplyvňuje miesto uloženia návratovej hodnoty a teda aj nutnosť použitia RVO.
- V prípade, že návratový typ funkcie má menšiu alebo rovnakú veľkosť ako register v ktorom má byť uložená návratová hodnota nie je nutné použiť RVO.
- Čísla s pohyblivou desatinnou čiarkou môžu byť vrátené aj v registroch FPU.
- Jazyk C++ obsahuje funkciu `std::move`, ktorá označuje použitie takzvanej *move sémantiky*. Indikuje že objekt môže byť uvoľnený a jeho zdroje môžu byť použité inak. Pri jej použití na návratovú hodnotu, môže nastať RVO.
- Štandard jazyka C++ určuje aj ďalšie prípady, v ktorých môže RVO nastať a kde určite nastane. Kompletný zoznam prípadov kedy nastane je teda závislý od implementácie prekladaču.

Implementácia tejto detekcia však vyžaduje ďalšie podrobné skúmanie štandardu C++, konvencií volania a správania jednotlivých prekladačov. Analyzované musia byť definície funkcií, nestačí analýza vstupov. Aj napriek zložitej analýze by tieto informácie vo väčšine prípadov pomohli len minimálne. Hlavné využitie RVO je pri predávaní veľkých štruktúr a triednych dátových typov, s ktorými RetDec momentálne nedokáže pracovať. Z týchto dôvodov bolo rozhodnuté, že analýza v rámci práce nebude implementovaná.

5.5 Návrh rozhrania knižnice demangleru

5.5.1 Rozhranie

Demangler potrebuje jednotné rozhranie. To je zabezpečené pomocou abstraktnej triedy. Táto trieda bude môcť byť jednoducho doplnená o rozhranie pre získavanie ďalších informácií o demanglovaných funkciách pre rekonštrukciu dátových typov.



Obr. 5.1: Abstraktná trieda Demangler predstavujúca rozhranie demangleru.

`demangleToString(mangled)`
Je abstraktná metóda používaná pre účely demanglingu do textových reťazcov. Jej vstupom je manglovaný symbol dátového typu `std::string`. V prípade úspešného priebehu demanglingu bude jej výstup demanglovaný reťazec taktiež dátového typu `std::string`. V prípade neúspechu pri demanglovaní bude vrátený prázdny reťazec. Dátové typy vstupov a výstupov sú rovnaké ako v pôvodnom demangleri. To zabezpečí jednoduchú migráciu na novú knižnicu.

`status()`
Je metóda, ktorou je kontrolovaný úspech demanglingu. Pomocou jej výstupu je možné skontrolovať dôvod neúspechu demanglingu.

5.5.2 Prístup k demangleru

Demangler bude dostupný rovnako ako v pôvodnej implementácií cez demangler provider. Pri štarte modulu `bin2llvmir` je demangler provider, rovnako ako všetky ostatné, inicializovaný pred spustením analýz. V tejto fáze už dekompilátor pozná (resp. odhadne) kompilátor použitý pre vytvorenie práve analyzovaného súboru. Táto informácia je dostupná v konfigurácií. Demangler provider obsahuje dve funkcie, pomocou ktorých je možné demangler vytvoriť a neskôr k nemu pristupovať.

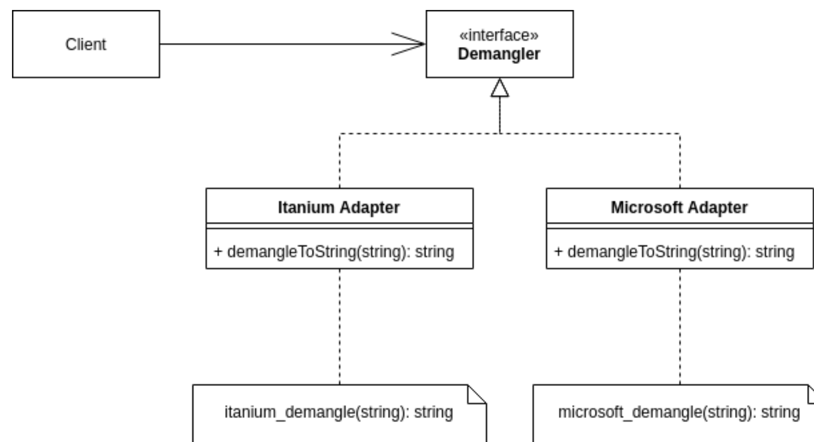
- `addDemangler(config)`
  Jej úlohou je vytvoriť demangler na základe informácie o použitom prekladači uloženej v konfigurácií.

- `getDemangler()`
  Je funkcia slúžiaca na získavanie demangleru. Všetky analýzy budú pristupovať k demangleru len cez túto funkciu.

5.5.3 LLVM demangler

Prepojenie

Pre použitie knižnice LLVM demangleru je potrebné napojiť ju na jednotné rozhranie v novonavrhovanom demangleri. Pre tieto účely by bolo možné využiť návrhový vzor adaptér a vytvoriť triedu, ktorá implementuje rozhranie demangleru. Metóda tejto triedy bude volať funkciu `itanium_demangle()` resp. `microsoft_demangle()` demanglovacej knižnice, ktorá vráti demanglované symboly v podobe reťazca znakov. V prípade nesprávneho manglovaného mena bude vrátený prázdny reťazec.



Obr. 5.2: Jednoduché prepojenie knižnice LLVM demangleru na navrhovanú knižnicu.

Kapitola 6

Implementácia knižnice demangleru a rozšírení analýzy parametrov

6.1 Demangler

6.1.1 LLVM Demangler

Využitie knižnice LLVM demangleru pre potreby dekompilátoru vyžadovalo knižnicu upraviť. Pri úpravách bola snaha aby zmeny neovplyvnili štruktúru programu. Väčšina úprav bolo špecifických pre jej použitie v dekompilátore a nemohli byť zavedené do upstreamu, ale ďalší vývoj knižnice a následná aplikácia úprav do verzie v dekompilátore bude veľmi jednoduchá.

Prístup k syntaktickému stromu

Spracovanie syntaktického stromu vyžadovalo aby funkcia demangleru vrátila ukazateľ na skonštruovaný strom a aby tento strom bol prístupný aj po ukončení demanglingu. Oba demanglery používajú vlastný alokátor z dôvodu zvýšenia rýchlosti. Ich implementácia je založená na alokovaní si určitého miesta a jeho následného využívania pre uloženie syntaktického stromu. Riešením tohto problému bolo vyňatie alokácie pred volanie demanglovacej funkcie a predanie referencie na alokátor do demanglovacej funkcie. To zaručilo že alokované miesto nebude uvoľnené hneď po ukončení demanglingu a je bezpečné pristupovať k syntaktickému stromu.

Assert

Pri testovaní Microsoft demanglingu boli objavené prípady, kedy pri neočakávanom vstupe (nevalidné manglované meno podľa Microsoft schémy) sa aplikácia dostala do stavu, ktorý spôsoboval pád aplikácie. Spôsobovala to kontrola neplatných stavov pomocou funkcie assert. Toto riešenie má negatívne následky v podobe potenciálneho násilného ukončenia aplikácie alebo octitnutia sa v neočakávanom stave. Túto chybu bolo ale bolo možné pomerne priamočiaro opraviť.

Relokácia reťazcov

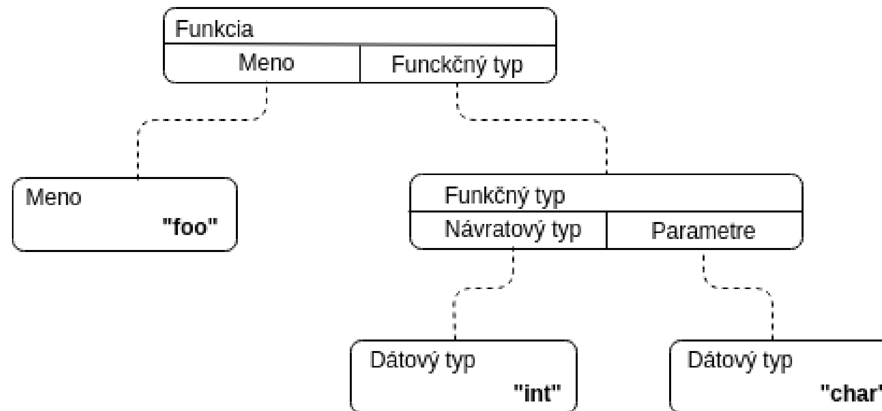
Ďalší potenciálny problém súvisel so spôsobom uloženia reťazcov v syntaktickom strome demangleru Itanium. Pri pokuse o jeho ďalšie spracovanie hrozil prístup na neplatnú adresu, čo by spôsobilo čítanie z nealokovanej pamäte. Tento problém bol špecifický pre spôsob použitia knižnice vopred neočakávaným spôsobom. Pre odstránenie problému bolo potrebné upraviť alokátor používaný knižnicou a do alokovaného priestoru kopírovať všetky reťazce, ktoré by potenciálne mohli spôsobiť problémy.

6.1.2 Borland demangler

Podobne ako v knižnici LLVM demangleru, bude na reprezentáciu demanglovaných symbolov použitá štruktúra syntaktického stromu. Narozdiel od LLVM demangleru, ale budú rozlišované uzly obsahujúce mená, dátové typy a všeobecne bude rozlišovaných čo najviac informácií, ktoré by mohli pomôcť pri ďalšom použití. Toto uľahčí ďalšie spracovanie. Demangler pre Borland taktiež používa jednotné rozhranie navrhnuté v kapitole 5. Hlavná časť demanglingu bude vykonávaná triedou `BoralndASTParser`, ktorá postupne vytvára stromovú štruktúru reprezentujúcu demanglovaný objekt. Pre demangling je využívaná metóda rekurzívneho zostupu.

Prevod syntaktického stromu do textovej reprezentácie

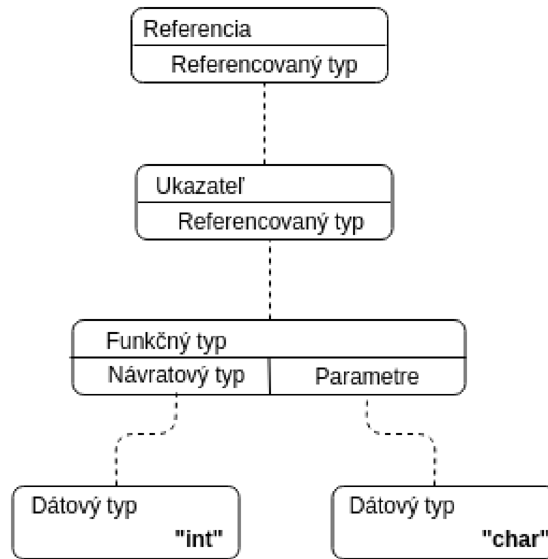
Jednotlivé uzly sú reprezentované objektami tried implementujúcich abstraktnú triedu `Node`. Každý uzol typ uzlu implementuje vlastnú virtuálnu funkciu `print()`, ktorá skladá textovú reprezentáciu celého podstromu aktuálneho uzla. Pre získanie textovej reprezentácie celého demanglovaného symbolu je funkcia volaná na koreň syntaktického stromu. Následne je rekurzívne volaná pre všetky poduzly. Pri prevode abstraktného stromu do textovej podoby bolo možné zvoliť rôzne cieľové jazyky. Napriek tomu, že zdrojové symboly môžu pochádzať z kompilácie jazyka Delphi, výstpné deklarácie majú podobu jazyka C++. Bolo to z dôvodu zachovania konzistentnej podoby výstupov rôznych demanglerov. V budúcnosti však môže byť relatívne jednoducho doimplementovaná podpora pre výstup do jazyka Delphi a aj iných. Voľba výstupu však komplikoval implementáciu.



Obr. 6.1: Príklad ukazujúci abstraktný syntaktický strom funkcie `int foo(char)` a jeho textovú reprezentáciu.

Problém vznikajúci pri vytváraní deklarácií je demonštrovaný na jednoduchom príklade deklarácie funkcie `foo`, ktorá vracia dátový typ `int` a má jeden parameter dátového typu `char`. Je spôsobený tým, že pri tvorbe AST bola snaha minimalizovať redundantné informácie. Deklarácia sa skladá z mena funkcie a typu funkcie. Typ funkcie je zložený z dátového typu návratovej hodnoty a usporiadanej množiny typov parametrov.[1] Pri textovej reprezentácii musí byť meno funkcie vnorené do deklarácie typu funkcie. Preto nie je možné najprv vytvoriť textovú reprezentáciu funkčného typu a potom mena, resp. naopak. Knižnica toto rieši rozdelením generácie výstupu na dve časti. Je možné najprv vytvoriť reprezentáciu návratového typu, potom mena funkcie a nakoniec dokončiť generáciu funkčného typu pridaním parametrov. Generácia výstupu je rozdelená na dve čatsi len pri uzloch, kde to dáva zmysel. Sú to konkrétne uzly reprezentujúce funkčné typy a polia. Polia sú deklarované podobným spôsobom ako funkcie, teda meno premennej rozdeľuje deklaráciu. Toto však pri demanglingu nikdy nenastane, pretože mená premenných nie sú uchovávané.

[1] Skutočná implementácia reprezentácie funkčného typu zahŕňa aj konvenciu volania a informáciu o tom či je funkcia variadická.



Obr. 6.2: Demonoštrácia viacnásobného vnorenia sa do deklarácie funkčného typu.

Spôsob deklarácie funkčných typov a polí ďalej komplikuje aj generovanie ukazateľov, L-value referencií a R-value referencí na tieto typy. Tento problém ilustruje obrázok 6.2. Typy, ktoré môžu byť rozdelené (pole a funkčný typ) obsahujú informáciu, že majú aj pravú sranu. Pri vytváraní ukazateľov a referencí sa z referencovaného typu do vytváraného typu propaguje informácia o tom či spomínanú pravú stranu majú alebo nie. Na základe tejto informácie je generácia textovej preprezentácia podstromov rozdelená na dva prípady.

6.2 Rekonštrukcia dátových typov

6.2.1 Konverzia syntaktického stromu do ctypes

Výstupom demanglingu je abstraktný syntaktický strom. Tento strom je prevádzaný do štruktúr knižnice `ctypes` pomocou ďalšej knižnice `astCtypesParser`. Pre každú knižnicu demangleru a teda pre tri formáty syntaktických stromov existuje samostatná implementácia. Ich spôsob práce je ale rovnaký. Pre konverziu sa využívajú len syntaktické stromy reprezentujúce funkciu. Z neho je získané meno funkcie. Dátové typy parametrov sa musia zisťovať postupným porovnávaním textovej reprezentácie typu (prípad demangleru pre Itanium) alebo v lepšom prípade porovnávaním s výčtovým typom (demangler Microsoft). V prípade demangleru Borland bolo počítané s touto nutnosťou už pri návrhu, preto sú všetky informácie usporiadané tak, aby bola konverzia čo najjednoduchšia.

6.2.2 Konverzia ctypes do LLVM IR

Pre účely konverzie štruktúr ctypes do LLVM IR sa v RetDecu už nachádza knižnica. Tá musela byť upravená aby ju bolo možné použiť aj pre potreby konverzie ctypes štruktúr vytvorených pri demanglingu. Pribudla konverzia nových dátových typov.

Referencie

Pri analýze a výbere vhodnej reprezentácie referencií v LLVM IR bol využitý prekladač Clang, ktorý ponúka možnosť kompilácie do LLVM IR. Kompiláciou funkcií s parametrami typu L-value referencia bolo zistené, že výsledný kód LLVM IR má rovnakú podobu, ako pri kompilácií funkcií s parametrami typu ukazateľ. Parameter typu L-value referencia mal však naviac atribút `dereferenceable(<n>)`, kde `<n>` je nahradené veľkosťou ukazateľa v bytoch. Tento úkaz je demonštrovaný na príklade 6.3.

```
; Function Attrs: nounwind sspstrong uwtable
define void @_Z7foo_refRi(i32* dereferenceable(4)) #0 {
    %2 = alloca i32*, align 8
    store i32* %0, i32** %2, align 8
    %3 = load i32*, i32** %2, align 8
    store i32 5, i32* %3, align 4
    ret void
}

; Function Attrs: nounwind sspstrong uwtable
define void @_Z7foo_ptrPi(i32*) #0 {
    %2 = alloca i32*, align 8
    store i32* %0, i32** %2, align 8
    %3 = load i32*, i32** %2, align 8
    store i32 5, i32* %3, align 4
    ret void
}
```

Obr. 6.3: Výstup pri kompilácií do LLVM funkcie s jedným parametrom typu ukazateľ na int (hore) a funkcie s jedným parametrom typu referencia na int (dole).

Je to atribút parametru funkcie, nie dátového typu. Preto nie je možné tento atribút nastaviť už pri konverzií dátového typu. Tento atribút musí byť nastavený až pri finálnej rekonštrukcií funkcie. Implementácie konverzie referencií prebieha teda rovnako ako konverzia ukazateľov. Najprv je vytvorený referencovaný dátový typ a potom ukazateľ, ktorý sa naň odkazuje. Knižnica Ctypes nerozlišuje L-value a R-value referencie. Je to z dôvodu, že pri analýze LLVM IR vzniknutého ich kompiláciou neboli nájdené žiadne rozdiely v ich reprezentácií.

Konverzia pomenovaných dátových typov

Musela pribudnúť aj konverzia pomenovaných dátových typov. Prebieha tak, že je vytvorený predvolený dátový typ, teda integer s veľkostou ukazateľa podľa architektúry. Táto konverzia je aktuálne jediná možnosť z dôvodu nepodpory spracovania zložitejších dátových typov na strane RetDecu. Pridanie podpory pre triedne dátové typy a štruktúry je však náročné a vyžadovalo by rozsiahle zmeny v celej štruktúre dekompilátoru. V prípade pridania tejto podpory by bolo možné na základe mena vytvoriť LLVM IR reprezentáciu dopredu známych dátových typov zo štandartných knižníc jazykov C++ a Delphi ako napríklad `std::string` alebo `std::vector`. Podobne aj v prípade pridania rekonštrukcie užívateľsky definovaných tried, by bolo možné relatívne jednoducho rozšíriť implementáciu o konverziu týchto tried do LLVM IR na základe mena získaného demanglingom.

6.2.3 Rozšírenie analýzy

Všetky modifikácie pre zlepšenie detekcie parametrov a návratových typov funkcií boli nakoniec použité v LLVM prechode `param_return`. Analýzu bolo nutné doplniť na viacerých miestach. Prvé miesto bola časť starajúca sa hlavne o detekciu štandardných funkcií jazyka C ako sú `printf()`. Tu sú doplňované informácie o funkciách štandardnej knižnice jazyka C++. Ďalšie miesto bola časť starajúca sa o obalovacie funkcie. To sú také funkcie, ktoré len zavolajú inú funkciu a predajú jej svoje argumenty. Posledný prípad zastrešuje všetky ďalšie možnosti. Porovnania s výstupmi, tak ako sú popísané v návrhu v sekcií 5.4.1. Pre analýzu a porovnávnie sa používajú primárne dáta v podobe ctypes. Tieto štruktúry obsahujú viac informácií ako ich LLVM IR reprezentácia. Tá je až následne vkladaná do modulu na ktorom prebehnú ďalšie analýzy.

Kapitola 7

Testovanie

7.1 Demangling

Pre testovanie neboli použité testy pôvodnej knižnice demangleru. Obsahovali nesprávne výstupy a museli byť manuálne upravené. Knižnice demangleru LLVM aj GCC obsahovali testy pre svoj demangler a slúžili ako ďalší zdroj testov. Ďalej bol demangler testovaný na vstupoch, u ktorých bolo známe, že ich pôvodný demangler nesprávne demangloval alebo ich spustenie spôsobilo pád aplikácie. Takisto boli vytvorené testy nové. Počas vývoja demangleru Borland bola použitá technika Test Driven Development alebo TDD. Pred implementáciou každej novej funkčnej vlastnosti bol najprv napísaný test, ktorý túto funkčnosť overoval. To zaručovalo takmer 100% pokrytie kódu testami.

7.1.1 Porovnanie úspešnosti s pôvodnou knižnicou demangleru

Pri testovaní bola nová knižnica porovnaná so starou knižnicou demangleru. Výsledky tohto porovnania ukazuje tabuľka 7.2. Testovacia sada ukázala zásadné zlepšenie pri všetkých prekladačoch. Najzásadnejšie zlepšenie sa ukázalo pri demanglingu Microsoft symbolov.

Schéma	Úspech pôvodný	Úspech nový	Celkom	Zlepšenie
Itanium	64	159	166	57%
Microsoft	249	780	788	67%
Borland	69	149	151	52%

Tabuľka 7.1: Výsledky porovnania úspešnosti demanglovania novej knižnice so starou.

Microsoft

Vysoká miera zlepšenia je pravdepodobne spôsobená veľkým množstvom informácií vkladanej do Microsoft schémy. Nová verzia demangleru podporuje aj C++17, čo taktiež zvýšilo percento úspešnosti novej knižnice. Ďalšie zásadné zlepšenie je, že počas testovania nespôsobil žiadny vstup pád aplikácie.

Borland

Najvýraznejšie zlepšenie sa však objavilo pri demanglingu šablón. Keďže už pri manglingu vstavaných dátových typov jazyka Delphi sa používajú mená manglované ako šablóny s číselnými argumentami, ich podpora je dôležitá. Nový demangler nedokázal demanglovať

symboly vzniknuté použitím šablón, ktorých argument bol ukazateľ na funckiu alebo referencia na dátový člen triedy.

7.1.2 Porovnanie rýchlosti s pôvodnou knižnicou demangleru

Porovnanie rýchlosti knižníc bolo náročné z dôvodu vysokej miery neúspechu pri použití pôvodnej knižnice. Nová knižnica podporuje možnosti demanglovania zložitejších symbolov, čo spôsobuje zvýšenie komplexity programu. Preto je aj pri zvýšení doby vykonávania demanglingu ťažké vyvodiť definitívne závery o rýchlostiach knižníc.

Nová knižnica pre schému Borland ukladá reprezentáciu častí mena tak, aby mohli byť znovu použité. Pri deaktivácií tejto funkcionality trval beh programu s rovnakou množinou testov dvojnásobný čas ako pôvodná knižnica. Je ale potrebné zobrať do úvahy, že pri spustení tej istej testovacej sady pôvodnou knižnicou bol demangling neúspešný vo viac ako 50% testovaných prípadov. Pri použití ukladania vytváraných štruktúr pre následné znovupoužitie ale rýchlosť môže narásť viac ako 4 násobne. Toto zvýšenie rýchlosti závisí na tom, ako často sa počas jedného behu programu opakujú funkcie a pomenované dátové typy. Pri analýze programov, sa často vyskytuje napríklad dátový typ `std::string`, ktorý je definovaný ako `std::basic_string<char, std::char_traits<char>, std::allocator<char>>`. Manglovaním vznikne meno, ktoré obsahuje štyri šablóny. Jeho demangling môže zabrať relatívne veľa času. Nová knižnica ale musí tento typ demanglovať len raz a ďalej postačuje prehľadať uložené dátové typy a použiť už vytvorenú reprezentáciu objektu. To demangling urýchľuje a šetrí miesto v pamäti.

Schéma	Pôvodný	Nový bez ukladania	Nový s ukladaním
Borland	7,40	13,38	3,01

Tabuľka 7.2: Výsledky porovnania novej knižnice demnagleru so starou pri použití manglovacej schémy Borland.

Využitie pamäte bolo porovnávané na 30 náhodne vybraných testovacích vstupoch. Rozdiel v maximálnom využití pamäte knižnicami pri demanglingu jedného mena v rámci behu programu nie je výrazný. Nová knižnica využíva v priemere o 20% pamäte menej. Pri demanglingu väčšieho počtu Borland symbolov však využívanie pamäte rastie. Dôvodom je ukladanie štruktúr AST pre zrýchlenie procesu demanglingu.

7.2 Využitie typových informácií pri dekompilácií

Nasledujúca sekcia ukazuje štyri vybrané výsledky dekompilácie reálnych programov. Vybrané boli príklady ukazujúce ako úspešné použitie informácií z demanglingu, tak aj prípady, ktoré môžu byť v budúcnosti vylepšené.

```
// Address range: 0x3060 - 0x3066
int64_t function_3060(int64_t a1, int64_t a2, int64_t a3, int64_t a4) {
    // 0x3060
    return _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_();
}

// Address range: 0x3060 - 0x3066
int64_t function_3060(bool a1, int64_t * a2, int64_t * a3, int64_t * a4) {
    // 0x3060
    return _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_node_baseS0_RS_(
    a1, a2, a3, a4);
}
```

Obr. 7.1: Výstup pri použití dekompilátoru bez modifikácií (hore) a výstup pri použití dekompilátoru pre určenie typov parametrov funkcie (dole).

Na príklade 7.1 je možné vidieť využitie informácií z demangleru pre detekciu typov parametrov. V pôvodnej verzií je možné vidieť, že analýza neodhalila žiadne parametre funkcia detekované parametre aj napriek tomu, že funkcia má mať parametre štyri. Tieto parametre boli pridané, čo sa odzrkadlilo na úspešnej detekcií, že funkcia je wrapper. To následne spôsobilo že vo výstupe mohola byť volaná funkcia vnútorná a nie wrapper.

```
// Address range: 0xa3ec - 0xa415
// Demangled: RipSniffer::RipSniffer(char const*)
int64_t _ZN10RipSnifferC1EPKc(int64_t * a1, int64_t a2) {
    int64_t result = (int64_t)a1;
    *a1 = (int64_t)"udp port 520 or udp port 521";
    *(int64_t *)(result + 8) = a2;
    return result;
}

// Address range: 0xa3ec - 0xa415
// Demangled: RipSniffer::RipSniffer(char const*)
int64_t _ZN10RipSnifferC1EPKc(int64_t * a1, char * a2) {
    int64_t result = (int64_t)a1;
    *a1 = (int64_t)"udp port 520 or udp port 521";
    *(int64_t *)(result + 8) = (int64_t)a2;
    return result;
}
```

Obr. 7.2: Výstup pri použití dekompilátoru bez modifikácií (hore) a výstup pri použití dekompilátoru pre určenie typov parametrov funkcie (dole).

Na príklade 7.2 je možné vidieť konštruktor triedy `RipSniffer`. Informácia o dátovom type explicitného parametru bola úspešne použitá, no nebolo detekované, že funkcia má aj skrytý parameter, ukazateľ na objekt. V prípadoch ako je konštruktor a deštruktor je možné z manglovaného mena zistiť že sa jedná a konštruktor resp. deštruktor a túto informáciu predať ďalej. To sa však zatiaľ nedeje a mohlo by byť predmetom ďalších rozšírení.

```
// Address range: 0x4087a4 - 0x408b28
int32_t _40_System_40_SysGetMem_24_qqri(void)

// Address range: 0x4087a4 - 0x408b28
// Demangled: __fastcall System::SysGetMem(int)
int32_t _40_System_40_SysGetMem_24_qqri(int32_t a1)
```

Obr. 7.3: Výstup pri použití dekompilátoru bez modifikácií (hore) a výstup pri použití dekompilátoru pre určenie typov parametrov funkcie (dole). Demangling odhalil parameter funkcie, ktorý predtým detekovaný nebol.

```
// Address range: 0x41bcc8 - 0x41bcce
int32_t _40_Amazon_40__40_GetPackageInfoTable_24_qqrv(void) {
    // 0x41bcc8
    g5 = &g44;
    return &g44;
}

// Address range: 0x41bcc8 - 0x41bcce
// Demangled: __fastcall Amazon::GetPackageInfoTable(void)
int32_t _40_Amazon_40__40_GetPackageInfoTable_24_qqrv(void) {
    // 0x41bcc8
    g5 = &g44;
    return &g44;
}
```

Obr. 7.4: Výstup pri použití dekompilátoru bez modifikácií (hore) a výstup pri použití dekompilátoru pre určenie typov parametrov funkcie (dole). Demangling neodhalil implicitný parameter a zlepšenie dekompilácie sa neprejavilo.

Príklady 7.3 a 7.4 ukazujú dekompiláciu emailovej knižnice pôvodne v jazyku Delphi. Keďže sa jedná o knižnicu a funkcie neboli v binárnom súbore nikdy volané, analýza parametrov nebola úspešná. Informácie o počte a dátových typoch parametrov sú získané len demanglingom. V príklade 7.4 je možné vidieť, že samotný demangling nedokázal odhaliť implicitný parameter, ktorým je predávaný ukazateľ na objekt a teda výsledok dekompilácie nebol vylepšený. Jeho odhalenie bez analýzy volaní je náročné a RetDec to aktuálne nedokáže. Príklad 7.3 na druhej strane ukazuje hlavičku funkcie pri úspešnom využití informácií dostupných z demanglingu. Funkcia má podľa informácií z demangleru jeden parameter, čo sa odzrkadlilo aj v dekompilovanom výsledku.

Kapitola 8

Záver

V tejto práci boli priblížené nástroje používané pri reverznom inžinierstve. Z týchto nástrojov sa podrobnejšie venovala dekompilátoru RetDec. Bol vysvetlený pojem mangling, jeho využitie a výskyt. Mangling bol predvedený na schémach prekladačov využívajúcich Itanium C++ ABI, Microsoft a Embarcadero.

Cieľom tejto práce bolo zlepšenie výsledkov demanglingu nástroja RetDec. Preto bol, následne navrhnutý a implementovaný spôsob ako s manglovaných symbolov získať demanglované. Pre tento účel boli upravené existujúce riešenia pre schému Itanium a Microsoft a vytvorené nové pre prekladače Embarcadero. Úspešnosť demanglingu sa použitím novej knižnice výrazne zlepšila. Demangling je teraz na veľmi dobrej úrovni a zlyháva len v naozaj zložitých prípadoch, ktoré sa v reálnych programoch veľmi zriedka vyskytujú. Navyše použitie aktívne vyvíjanej knižnice LLVM pre demangling symbolov Itanium a Microsoft sľubuje vylepšovanie a opravu nedostatkov nájdených počas testov. Návrh demangleru pre Borland taktiež sľubuje pomerne jednoduché odstránenie nedostatkov a prípadné rozšírenia. V budúcnosti by mohla byť knižnica Borland demangleru zrýchlená používaním vlastného alokátoru podobne, ako je používaný v ďalších dvoch knižniciach. Ďalším spôsobom ako vylepšiť knižnicu by mohlo byť pridanie možnosti demanglingu ďalších jazykov. Kandidátom by mohol byť napríklad jazyk Swift, čo je to pomerne nový, kompilovaný jazyk používaný hlavne v zariadeniach Apple.

Ako ďalší cieľ si práca kládla využitie informácií získaných demanglingom pre zlepšenie výsledkov spätného prekladu spomínaného spätného prekladaču. Tento cieľ bol takisto úspešne splnený. Dátové typy získané demanglingom sú úspešne používané aj pri funkciách so skrytými parametrami. Hlavné slabiny sa vyskytujú pri nevyužívaní dátových typoch, s ktorými RetDec momentálne pracovať nevie, ako sú napríklad triedne dátové typy a referencie. Knižnica demangleru je ale pripravená aby v budúcnosti boli používané aj tieto typy. Implementované riešenie analýzy parametrov by mohlo byť v budúcnosti rozšírené o pokročilejšiu analýzu skrytých parametrov. Na jej základe by sa dalo častejšie presne určiť, že sa jedná o metódu triedy. Toto rozšírenie by mohlo byť použité pre prípadnú rekonštrukciu tried.

Literatúra

[1] Programming languages – C++. Technická Správa ISO/IEC 14882:2017, International Organization for Standardization, 2017.
URL https://www.iso.org/standard/68564.html

[2] C++ Reference. 2019.
URL http://docwiki.embarcadero.com/RADStudio/Tokyo/en/C%2B%2B_Reference

[3] Clang 9 documentation. 2019, navštívené 29.4.2019.
URL https://clang.llvm.org/docs/

[4] Delphi Language Reference. 2019, navštívené 29.4.2019.
URL http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Delphi_Language_Reference

[5] GCC online documentation. 2019, navštívené 29.4.2019.
URL https://gcc.gnu.org/onlinedocs/

[6] Chikofsky, E. J.; Cross, J. H.: Reverse engineering and design recovery: a taxonomy. *IEEE Software*, ročník 7, č. 1, jan 1990: s. 13–17, doi:10.1109/52.43044.

[7] CodeSourcery; Compaq; EDG; aj.: Itanium C++ ABI. Navštívené 20.12.2018.
URL https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[8] Eilam, E.: *Reversing: secrets of reverse engineering*. John Wiley & Sons Inc, 2005, ISBN 0764574817.

[9] Fog, A.: Calling conventions for different C++ compilers and operating systems. 2018, navštívené 29.4.2019.
URL https://www.agner.org/optimize/calling_conventions.pdf

[10] Křoustek, J.; Matula, P.; Zemek, P.: RetDec: An Open-Source Machine-Code Decompiler. December 2017, technická správa, prezentované na konferencí Botconf 2017.

[11] Levine, J. R.: *Linkers and loaders*. Morgan Kaufmann, 2010.

[12] Michael Sikorski, A. H.: *Practical Malware Analysis*. Random House LCC US, 2012, ISBN 978-1593272906.

Príloha A

Príloha

A.1 Gramatika manglovacej schémy prekladačov Embacadero.

Gramatika je vo forme EBNF s nasledujúcim významom:

- neterminály sú vyznačne <>
- terminály sú značené "
- | označuje alternatívu
- [] označuje jeden z terminálov, pri použití s jedným terminálom znamená 0-1 opakovanie
- {} označuje ľubovolný počet opakovaní
- všetky pravidlá sú ukončené ;
- terminály a neterminály môžu byť spájané do blokov ohraničených zátvorkami ()
- komentáre sú ohraničené znakmi /* a */

```
<mangled-name> ::= <mangled-function> ;
<mangled-function> ::= <full-name> '$' <func-info> ;
<full-name> ::= '@' {<namespace>} <func-name> ;
<func-name> ::= <name> | <operator> ;
<name> ::= ['a'-'z''A'-'Z''0'-'9''_']+ | <template-name> ;
<namespace> ::= <func-name> '@' ;
<template-name> ::= '%' <name> '$' <template-args> '%' ;
<operator> ::= '$o' <type>
             | '$badd'  | '$bsubs' | '$bsub'  | '$basg'
             | '$bmul'  | '$bdiv'  | '$bmod'  | '$binc'
             | '$bdec'  | '$beql'  | '$bneq'  | '$bgtr'
             | '$blss'  | '$bgeq'  | '$bleq'  | '$bnot'
             | '$bland' | '$blor'  | '$bcmp'  | '$band'
             | '$bor'   | '$bxor'  | '$blsh'  | '$brsh'
             | '$brplu' | '$brmin' | '$brmul' | '$brdiv'
             | '$brmod' | '$brand' | '$bror'  | '$brxor'
             | '$brlsh' | '$brrsh' | '$bind'  | '$badr'
             | '$barow' | '$barwm' | '$bcall' | '$bcoma'
             | '$bnew'  | '$bnwa'  | '$bdele' | '$bdla'
             | '$bctr1' | '$bctr2' | '$bctr'  | '$bdtr1'
             | '$bdtr2' | '$bdtr' ;
<template-args> ::= <template-arg> {<template-arg>};
<template-arg> ::= <type> [<non-class-temlpate-arg>];
<non-class-temlpate-arg> ::= '$i' [-] <number>;
<func-info> ::= <qualifiers> <func-type> ;
<qualifiers> ::= ['w'] ['x'] ['r'];
                /* w - volatile, x - const, r - restricted */
<func-type> ::= <call-conv> <func-params>
                [<var-arg-params>] [$ <type> ];
<call-conv> ::= 'qqr' | 'qqs' | 'Q' | 'q';
                /* qqr - fast-call, qqs - stdcall, */
                /*   q - cdecl,       Q - pascal   */
<func-params> ::= <func-param> {<func-param>};
<func-param> ::= <type> | <backref>;
<backref> ::= 't' ['1'-'9''a'-'z'];
<var-arg-params> ::= 'e';
<type> ::= <qualifiers> (<pointer-type> | <ref-type>
         | <rref-type> | <array-type> | <func-type>
         | <named-type> | <built-in-type>);
<pointer-type> ::= 'p' <type>;
<ref-type> ::= 'r' <type>;
<rref-type> ::= 'h' <type>;
<array-type> ::= 'a' <type>;
<named-type> ::= <number> <name>;
<built-in-type> ::= 'o' | 'b' | 'Cs' | 'Ci' | 'v' | 'zc'
                  | 'uc' | 'c' | 'u'  | 's'  | 'i' | 'l'
                  | 'j'  | 'f' | 'd'  | 'g'  | 'N';
<number> ::= ['1'-'9'] {['0'-'9']};
```

Príloha B

Obsah přiloženého pameťového média

Súbor	Popis
README	Ďalšie podrobnosti o súboroch
doc/	Zložka s dokumentáciou
retdec/	Zložka so zdrojovými súbormi dekompilátoru
install/	Zložka so skompilovanými programmi
tests/	Zložka s testovacími príkladmi

Tabuľka B.1: Obsah priloženého pamäťového média.