VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

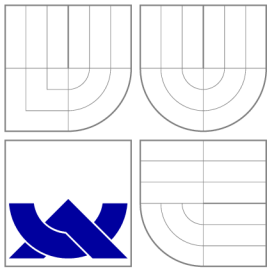
ZPĚTNÝ PŘEKLAD VYSOKOÚROVŇOVÝCH KONSTRUKCÍ JAZYKA C++

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

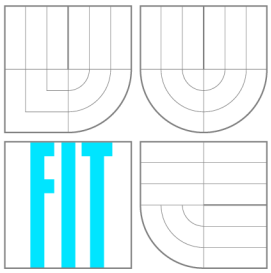
AUTOR PRÁCE
AUTHOR

Bc. DUŠAN JAKUB

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ZPĚTNÝ PŘEKLAD VYSOKOÚROVŇOVÝCH KONSTRUKCÍ JAZYKA C++

DECOMPILATION OF HIGH-LEVEL CONSTRUCTIONS IN C++ BINARIES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. DUŠAN JAKUB

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER MATULA

BRNO 2015

Zadání diplomové práce/17893/2014/xjakub19

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2014/2015

Zadání diplomové práce

Řešitel: **Jakub Dušan, Bc.**
Obor: Informační systémy
Téma: **Zpětný překlad vysokoúrovňových konstrukcí jazyka C++**
Decompilation of High-Level Constructions in C++ Binaries
Kategorie: Překladače

Pokyny:

1. Seznamte se s problematikou zpětného překladu binárního kódu do vyšší formy reprezentace.
2. Studujte zpětný překladač projektu Lissom. Zkoumejte kvalitu produkovaného kódu.
3. Analyzujte kód generovaný překladači jazyka C++. Zaměřte se jak na datové, tak řídicí konstrukce tohoto jazyka. Dále studujte existující metody zpětného překladu aplikací napsaných v C++.
4. Po konzultaci s vedoucím vyberte podmnožinu konstrukcí jazyka C++, na které se zaměříte. Následně navrhněte metody, které umožní jejich zpětný překlad. Řešení musí být nezávislé na konkrétní architektuře i překladači.
5. Metody navržené v předchozím bodě implementujte.
6. Vytvořené řešení důkladně otestujte sadou minimálně dvaceti testů, zohledněte různé procesorové architektury a překladače. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- Křoustek, J.: *Analýza a převod kódů do vyššího programovacího jazyka*, diplomová práce, FIT VUT v Brně, 2009.
- *Popis platformy LLVM* [online], 2012 [cit. 2012-10-01]. Dostupný z WWW: <www.llvm.org>.
- Skochinsky, I.: *Practical C++ decompilation*, RECON, 2011.
- Interní dokumentace projektu Lissom.
- Další dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Prvních čtyři body zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matula Peter, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2014
Datum odevzdání: 27. května 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Práce se zabývá dekompilací konstrukcí vysokoúrovňového objektového jazyka C++ ze strojového kódu. Je definován pojem zpětného překladu a popsány existující zpětné překladače s důrazem na dekompilaci C++. Dále je představen dekompilátor AVG, v jehož rámci tato práce vznikla. Je analyzován jazyk C++, a to jak na úrovni konstrukcí jazyka, tak na úrovni strojového kódu, a jsou představeny existující metody jeho dekompilace. Na jejich základě je navržen postup dekompilace tříd, jejich hierarchie, konstruktorů, destruktorů a virtuálních metod. Je detekováno i volání virtuálních metod. Navržený postup je implementován, podroben experimentům a zhodnocen. V závěru je nastíněno několik návrhů na další vývoj.

Abstract

The thesis addresses the decompilation of high-level object-oriented C++ language from a machine code. The term reverse engineering is defined and existing decompilers are described with emphasis on their ability to reconstruct C++. AVG decompiler project is introduced, to which this thesis contributes. C++ language is analysed, both on a logical level and in the machine code and existing methods of decompilation are described. On this basis a novel method is introduced, capable of decompiling classes, their hierarchy, constructors, destructors and definitions and usages of virtual methods. The method is implemented, tested and evaluated. In the conclusion, several suggestions for future development of this project are presented.

Klíčová slova

Zpětné inženýrství, dekompilace, strojový kód, assembler, C++, objektový jazyk, třída, dědičnost, tabulka virtuálních metod, konstruktor, destruktor, pozdní vazba, LLVM, AVG

Keywords

Reverse engineering, decompilation, machine code, assembler, C++, object-oriented language, class, inheritance, virtual method table, constructor, destructor, late binding, LLVM, AVG

Citace

Dušan Jakub: Zpětný překlad vysokoúrovňových konstrukcí jazyka C++, diplomová práce, Brno, FIT VUT v Brně, 2015

Zpětný překlad vysokoúrovňových konstrukcí jazyka C++

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Petera Matuly. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Dušan Jakub
24. května 2015

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Peteru Matulovi za příkladné vedení mojí práce a celému týmu překladače AVG za cenné rady. Dále děkuji svojí ženě Soni za podporu a povzbuzování během psaní celé práce.

© Dušan Jakub, 2015.
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

Seznam obrázků

Kapitola 1

Úvod

Počítače dnes prožívají obrovský rozmach ve spotřební elektronice. Otevřené operační systémy pronikají do oblastí, které byly ještě před nedávnou dobou zcela vyhrazeny systémům uzavřeným nebo jen jednoúčelovým firmwarům. Většina z nás používá hned několik takových přístrojů – mobily, tablety, chytré televize a podobně.

Není to tak dávno, co drtivá většina počítačů obsahovala procesor architektury x86 a na disku měla nainstalovanou jednu z verzí operačního systému Windows. To dnes již neplatí – podíl stolních počítačů klesá, zatímco mobilní chytrá zařízení jsou na vzestupu. Hardwarová i softwarová výbava těchto zařízení je mnohem rozmanitější.

Všechna tato zařízení umožňují instalaci a spouštění programů třetích stran. Uživatelé tak dostávají do ruky nástroj, který si mohou dovybavit podle svých potřeb. Ne vždy ale nainstalované programy dělají to a pouze to, co od nich uživatelé chtějí. Autoři počítačových virů brzy objevili nové pole působnosti a ochrana před nimi přichází pomalu. Antivirové společnosti mají své produkty i podpůrný software často úzce specializovaný na jednu architekturu a přizpůsobení je náročný proces.

Zpětný překladač vyvíjený antivirovou společností AVG Technologies, s.r.o., má za jeden z cílů pomoci s analýzou virů nezávisle na tom, zda se jedná o vir určený pro počítač nebo mobilní zařízení.

Jak se počítače zrychlují a přibývá paměti i místa na disku, programy a jejich autoři si mohou dovolit luxus vyššího programovacího jazyka i pro výpočetně náročné úlohy. I mezi viry se tak začínají objevovat jazyky, jakým je například C++. Cílem této práce je detekovat a analyzovat některé konstrukce tohoto jazyka tak, aby byla analýza vzniklého kódu co nejsnazší.

Práce je strukturovaná následovně: Kapitola 2 definuje základní pojmy v problematice zpětného inženýrství a jsou zde popsány existující zpětné překladače. Kapitola 3 popisuje, jak funguje zpětný překladač AVG. V kapitole 4 je představen jazyk C++ na logické úrovni. Následuje kapitola 5 popisující jeho překlad do strojového kódu a existující metody jeho zpětného překladu. Návrh mého řešení je pak načrtnut v kapitole 6. Dále následuje kapitola 7, která popisuje implementaci tohoto řešení pro zpětný překladač AVG. S hotovým řešením je experimentováno v kapitole 8, kde jsou také určena jeho silná a slabá místa. V závěrečné 9. kapitole je celá práce zhodnocena a je vytyčen směr pro další rozvoj.

Kapitola 2

Zpětné inženýrství v informatice

Reverzní inženýrství [19, 26] (též někdy nazýváno zpětné inženýrství, anglicky reverse engineering) je proces, pomocí něhož získáváme z hotového produktu informace o jeho specifikaci, návrhu či postupu výroby. Pojem zpětné inženýrství se netýká pouze počítačových programů. Naopak největšího vzestupu dosáhl s rozmachem průmyslu, kdy vnitřní struktura výrobků přestávala být zřejmá na první pohled a pro její pochopení bylo třeba využít důkladný výzkum.

V kontextu informačních technologií myslíme pojmem zpětné inženýrství především získávání zdrojového kódu (vyšší či nižší formy reprezentace) z přeloženého spustitelného programu. V tomto smyslu bude pojem používán i dále v této práci.

Získat zdrojové kódy pomocí zpětného inženýrství je samozřejmě náročné, nabízí se tedy otázka, proč se o to pokoušet. Existuje mnoho důvodů a řada z nich je naprosto legitimních. Některé z nich mohou být považovány za morálně závadné, za to však nenese odpovědnost technologie samotná, nýbrž ti, kteří ji takto používají.

Některé důvody k využití shrnuje následující seznam, nelegitimní důvody jsou označeny vykřičníkem:

- Získání ztracených zdrojových kódů;
- Migrace programu již nepodporovaného výrobcem na novou platformu;
- Bezpečnostní audit softwaru třetích stran;
- Hledání zranitelností za účelem jejich zneužití (!);
- Cracking – odstranění ochrany softwaru proti krádeži (!);
- Virová analýza.

Právě poslední bod, analýza virů, je důvodem, proč vznikl zpětný překladač AVG a v rámci něj i tato práce.

2.1 Nástroje reverzního inženýrství

Softwarových nástrojů asistujících při reverzním inženýrství existuje celá řada. Liší se tím, do jaké míry jsou schopny zdrojový kód rekonstruovat.

2.1.1 Disassembler

Disassembler (někdy též zpětný assembler) je nástroj, který provádí zpětný překlad ze strojového kódu dané platformy do jazyka symbolických instrukcí. Jedná se o přímý protiklad k assembleru, který překládá z jazyka symbolických intrukcí do strojového kódu.

Oba nástroje jsou algoritmicky poměrně jednoduché – existuje jednoznačná převodní tabulka mezi instrukcí strojového kódu (včetně všech parametrů) a její binární reprezentací. Přesto je nutné si uvědomit, že mnoho informací i z takto nízkoúrovňového zdrojového kódu je dopředným překladem ztraceno. Jedná se například o:

- Komentáře, formátování;
- Názvy návěstí a paměťových míst;
- Makra (jsou vždy rozbalena na všech místech použití);
- Idiomatické instrukce – některé assemblery používají kvůli lepší čitelnosti různá jména (aliasy) pro stejnou binární instrukci.

Assembler i disassembler je obvykle určen pro jednu konkrétní platformu. Každá platforma má svou vlastní sadu instrukcí, které jsou prováděny procesorem, sadu registrů, konvence apod. Stejně tak se liší i jazyk symbolických instrukcí pro každou z nich.

Pravděpodobně nejrozšířenějším univerzálním disassemblerem je IDA – Interactive Disassembler [6]. Poslední verze tohoto nástroje je pouze komerční, limitovanou edici starší verze je však možné stáhnout z oficiálních stránek zdarma pro osobní použití. Zdrojové kódy žádné verze tohoto disassembleru nejsou k dispozici.

2.1.2 Zpětný překladač do procedurálního jazyka

Zatímco disassembler cílí pouze na jazyk symbolických instrukcí, zpětný překladač převádí vstupní strojový kód do vyššího programovacího jazyka. Často využívá disassembler jako jeden z prvních stupňů své činnosti.

Zpětný překlad do vyššího jazyka je mnohonásobně náročnější než disassemblování. Zpětný překladač musí umět identifikovat řadu vlastností, které se ve zdrojovém kódu nenacházejí. Zatímco u disassembleru se jednalo o věci víceméně podružné, které pouze usnadňovaly člověku pochopit vygenerovaný kód, zde je nutné správně identifikovat prvky, které jsou nepostradatelné pro cílový jazyk. Jedná se například o:

- Funkce – hlavičky i těla;
- Jednoduché datové typy;
- Složené datové typy – struktury, kolekce a jejich kombinace;
- Volání funkcí;
- Podmínky a cykly.

Pro úspěšnou dekompilaci bývá výhodné znát nejen zdrojovou platformu, ale také jazyk, ve kterém byl původní program napsán, a překladač (včetně verze), který byl použit pro kompilaci. Tyto informace lze do určité míry zjistit i přímo ze strojového kódu a používají se pro přesnější dekompilaci např. idiomů – posloupností instrukcí, kterými překladač nahrazuje určitou operaci. Nezávisle na jazyku původního programu bývá výstupním jazykem dekompilace často C. Těchto překladačů existuje celá řada, detailně se jim bude věnovat kapitola 2.2.

2.1.3 Zpětný překladač do objektového jazyka

Objektové jazyky přinášejí další vrstvu abstrakce, je tedy opět obtížnější je správně dekompilovat.

Mezi informace, které chceme zjistit, patří například:

- Třídy a třídní hierarchie;
- Virtuální a nevirtuální členské metody;
- Určení polymorfních datových typů.

Zde již ale velmi záleží na jazyku. Jazyk C++, který je tématem této práce, dekompilaci příliš neusnadňuje, jelikož ve strojovém kódu je minimum metainformací, které by šlo použít.

2.2 Existující zpětné překladače

Dekompilátor AVG není jediným ani prvním zpětným překladačem. V této podkapitole popíši některé další dekompilátory a především zhodnotím jejich schopnost dekompilovat jazyk C++.

2.2.1 Hex-Rays

Dekompilátor Hex-Rays [7] je pravděpodobně nejpopulárnější zpětný překladač současnosti. Začínal jako doplněk do disassembleru IDA, dnes je disassembler i dekompilátor prodáván dohromady pod názvem Hex-Rays jako vlajkový produkt společnosti. Umožňuje dekompilovat do pseudokódu podobného C. Existuje ve třech verzích podle platformy dekompilovaného programu: x86, x64 a ARM (nedá se tedy mluvit o rekonfigurovatelnosti).

Hex-Rays umožňuje dekompilaci plně automatickou i interaktivní. V interaktivní dekompilaci uživatel prochází vygenerovaný kód a přiřazuje informace, jako jsou názvy a typy proměnných či definice datových typů, podle kterých se kód průběžně přegenerovává.

Podpora C++ je minimální: Hex-Rays například nalezne automaticky volání operátoru new (ve skutečnosti knihovní funkce), ale třídy, jejich atributy, třídní metody a tabulky virtuálních funkcí je nutno najít ručně. Výjimky nejsou podporovány.

Pro Hex-Rays, resp. IDA existují i doplňky třetích stran, které přidávají částečnou podporu dekompilace C++. Zmiňme vtbl-ida-pro-plugin [15], který analyzuje virtuální tabulky, a IDA ClassInformer PlugIn [5], který zpracovává RTTI informace. Žádný z nich ale není příliš propracovaný, oba se soustředí pouze na jeden typ analýzy a působí spíše dojmem experimentu než podporovaného řešení.

2.2.2 Boomerang

Boomerang [2] je open-source dekompilátor, jedná se pravděpodobně o první pokus o rekonfigurovatelný zpětný překladač. Rekonfigurovatelnost ale není úplná, v současnosti je podporována pouze architektura x86 (ELF, PE), SPARC (Solaris) a PowerPC (ELF, Mach-O) a přidání další architektury je „mnohaměsíční práce“ [3].

Překlad probíhá automatizovaně v několika fázích, ve kterých jsou postupně rekonstruovány jednotlivé konstrukce vyššího programovacího jazyka. Výstupním jazykem je C.

Jeho velkou nevýhodou je neschopnost detekce staticky linkovaných funkcí, výsledný kód tak obsahuje mnoho kódu ze systémových knihoven a většinou bez manuální úpravy není přeložitelný.

Podpora C++ je prakticky nulová, Boomerang dokáže pouze nalézt implicitně předaný ukazatel na this pomocí konvence `__thiscall`.

Další nevýhodou je, že k datu psaní této práce se zdá, že vývoj uvízl na mrtvém bodě.

2.2.3 Dcc

Experimentální zpětný překladač dcc byl vyvinut Cristinou Cifuentes v roce 1994 jako součást její dizertační práce [18]. Dekompiluje pouze základní konstrukce jazyka C a podporuje jen spustitelné soubory pro operační systém DOS a architekturu x86.

Podpora C++ není žádná a zmiňuji ho proto, že se jedná a jeden z prvních open source dekompilátorů.

2.2.4 REC Studio 4 – Reverse Engineering Compiler

REC [12] je decompiler šířený zadarmo, nikoli však s otevřeným zdrojovým kódem. Deklaruje podporu pro architektury x86, x64, Mips, PowerPC a mc68k, ARM je plánována. Výstupem je upravené C.

REC vyniká v dekompilaci programů zkompilovaných s informacemi pro debugger (ve formátech Dwarf2, PDB). Z těchto informací dokáže rekonstruovat i názvy C++ tříd a jejich vztahy. Naopak programy bez těchto informací jsou dekompilovány podstatně hůře. Problém je například s automatickou rekonstrukcí typů, které jsou dosazeny pouze, je-li detekováno volání systémové funkce se známou signaturou.

2.2.5 SmartDec

SmartDec [13] je komerční produkt, na stránkách projektu lze stáhnout demoverzi. Může fungovat samostatně nebo jako doplněk do disassembleru IDA.

Jeho výstupním jazykem je opět C. Podporovány jsou programy architektur x86 a x64 ve formátu ELF a PE, v případě kombinace s IDA i další objektové formáty.

Autoři deklarují podporu i pro C++. Má být detekována třídní hierarchie a výjimky.

2.2.6 Shrnutí

Jak je vidět z tabulky 2.1 i předchozího popisu, podpora dekompilace C++ není v současnosti běžná. REC Studio využívá pouze informace debuggeru, které většinou nejsou dostupné a nelze na ně spoléhat. Jediný SmartDec deklaruje podporu pro dostatečnou část konstrukcí jazyka C++, tento dekompilátor však není rekonfigurovatelný – podporuje jen x86 a x64.

	HexRays	Boomerang	dcc	REC Studio 4	SmartDec
Podporované architektury	x86, x64, ARM	x86, Sparc, PowerPC	x86	x86, x64, Mips, PowerPC, mc68k	x86, x64
Objektové formáty	cca 40 nejrozšířenějších	ELF, PE	MS-DOS	ELF, PE, Mach-O	ELF, PE / vše, co IDA
Licence	komerční	BFD/GPL	GPL	freeware	neznámá
DWARF	ano	ne	ne	ano	ne
PDB	ano	ne	ne	ano	ne
Interaktivní rozhraní	ano	ano	ne	ano	ne
Stále vyvíjen	ano	ne	ne	ano	ne
Podpora C++	Minimální	Minimální	Žádná	Třídní hierarchie z debugovacích informací	Třídní hierarchie, výjimky

Tabulka 2.1: Shrnutí existujících dekompilátorů

Kapitola 3

Zpětný překladač AVG

Dekompilátor AVG si klade za cíl plně rekonfigurovatelý zpětný překlad. Využívá faktu, že mnoho analýz je nezávislých na operačním systému, platformě či překladači. Snaží se abstrahovat od rozdílů a najít dostatečně univerzální vnitřní reprezentaci dekompilovaného kódu, takže teoreticky podporuje jakoukoli architekturu – je pouze nutno pro každou vytvořit formální popis.

Dekompilátor původně vznikl jako součást projektu Lissom [9] na Fakultě informačních technologií Vysokého učení technického v Brně jako jeden z nástrojů sdruženého vývoje hardware a software (hs/sw codesign). Postupně se ale osamostatnil a nyní jeho vývoj probíhá ve firmě AVG Technologies, s.r.o.

3.1 Použité technologie

Dekompilátor AVG nachází nové využití u dvou technologií, které byly původně vytvořeny ke zcela jinému účelu.

3.1.1 ISAC

Jazyk ISAC (Instruction Set Architecture C) vznikl v rámci projektu Lissom jako jazyk pro popis architektur. Umožňuje popsat jak strukturu (počet a typ registrů, paměťové adresování), tak chování (konání jednotlivých instrukcí) mikroprocesorové architektury. Je stanoveno jak binární kódovaní všech instrukcí, tak jejich přesné chování vhledem k registrům, paměti, atd.

Jeho původním účelem bylo usnadnění návrhu nového hardwaru a softwaru (hw/sw codesign), v překladači AVG je však použit pro dekódování instrukcí architektur stávajících.

3.1.2 LLVM IR

Jazyk LLVM IR [11] byl vyvinut jako jazyk interní reprezentace pro potřeby projektu LLVM [10]. Jedná se o nízkoúrovňový jazyk podobný assembleru, který je však naprosto platformně nezávislý.

Kód v LLVM IR se člení na funkce, základní bloky a instrukce. Naprostá většina instrukcí je tříadresných (2 operandy a 1 výsledek).

Typový systém zahrnuje jak jednoduché, tak odvozené typy. Z jednoduchých jsou podporovány celočíselné typy o libovolné bitové délce (což zahrnuje i znakové typy a typ boolean),

desetinné typy a prázdný typ void. Z odvozených pak struktury, pole, typované ukazatele a zvláštním typem jsou také funkce.

Chybí zde však explicitní podpora pro objektové datové typy a výjimky.

Silnou vlastností LLVM IR je Static Single Asignment (SSA). To značí, že do každé proměnné může být zapsána hodnota pouze jednou (například jako výsledek výrazu či výstup funkce). To přináší obrovskou výhodu při analýzách.

V případě, že definice proměnné překračuje hranice základního bloku, může se stát, že hodnota závisí na tom, ze kterého bloku bylo předáno řízení (například do proměnné je zapisováno v obou větvích podmínky if/else).V takovém případě musí být použita speciální instrukce PHINode, která hodnotu vybere.

Mezi další instrukce důležité pro následující analázy patří:

- `Load`, `Store`: Slouží k přístupu k paměti dané ukazatelem. V kódu generovaném AVG dekompilátorem jsou tyto instrukce použity nejen pro běžnou dereferenci ukazatelů, ale jsou jimi emulovány i měnitelné globální a lokální proměnné, a také přístup k registrům procesoru;
- Aritmetické a logické instrukce (`Add`, `Sub`, `And`, `Or`, ...);
- Porovnávací instrukce (`Cmp`);
- Intrukce přetypování: Je podporováno přetypování mezi různými typy ukazatelů (`BitCast`), mezi číslem a ukazatelem (`IntToPtr`, `PtrToInt`) a další;
- `GetElementPtr`: Instrukce pro získání ukazatele na prvek struktury nebo kolekce.
- Řídicí instrukce: Skoky (`Branch`), volání funkcí (`Call`), návrat z funkce (`Return`);

Jazyk LLVM IR slouží skvěle pro interní reprezentaci kódu pro dopředný překladač, a jak se ukazuje, stejně tak je vhodný pro překladač zpětný.

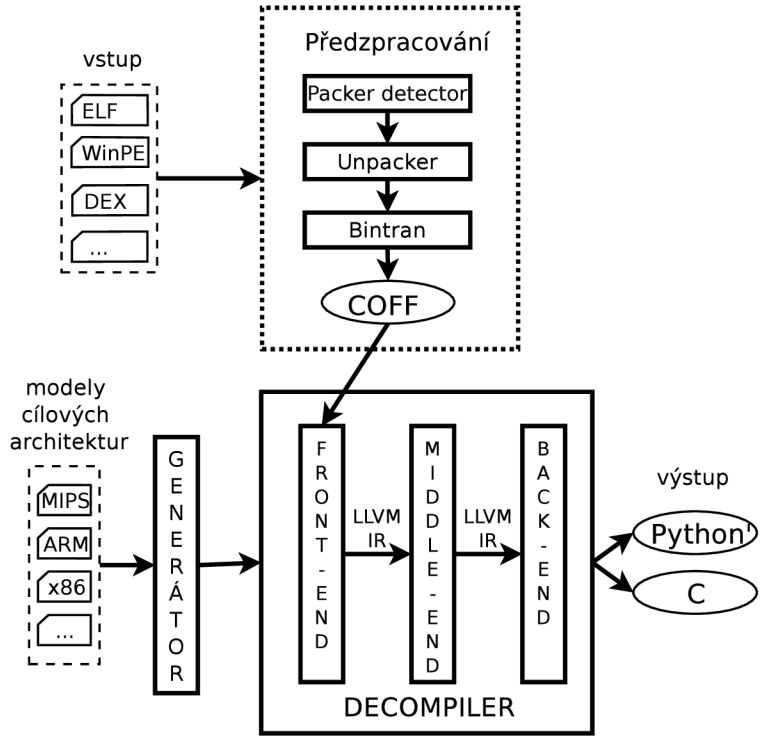
3.2 Vnitřní struktura

Vnitřní struktura dekompilátoru AVG [28, 23] je velmi podobná struktuře klasických překladačů. Existuje zde přední část (front end) zpracovávající vstupní kód, optimalizační střední část (middle end) a zadní část (back end), generující výsledný vysokoúrovňový kód. Navíc je zde nultá část provádějící předzpracování.

Schéma struktury je znázorněno na obrázku 3.1.

3.2.1 Předzpracování

Spustitelné soubory všech běžně používaných operačních systémů a platforem neobsahují pouze zkompilovaný strojový kód, nýbrž i další informace, kód navíc může být zkomprimovaný. Ve fázi předzpracování je ze vstupního souboru zjištěno co nejvíce informací o jeho formátu, platformě, jazyku, překladači a podobně. Vstupní, platformně závislý soubor je dekomprimován, je-li to třeba, a převeden do jednotného objektového formátu. Byl-li program zkompilován s informacemi pro ladění, jsou tato data dekódována a uložena pro další použití dekompilátorem. Nakonec je podle detekované platformy vytvořen dekodér instrukcí z binárního kódu.



Obrázek 3.1: Schéma dekompilátoru AVG, přejato z [28], upraveno

3.2.2 Přední část (front end)

Cílem přední části je dekódovat strojový kód, uložený již v jednotném objektovém formátu, a vytvořit kód v jazyku interní reprezentace LLVM IR.

Přední část k tomu využívá model zdrojové architektury v jazyce ISAC. Každá instrukce vstupního kódu je samostatně dekódována a přeložena do posloupnosti mikroinstrukcí vygenerovaných podle modelu v jazyce ISAC.

V přední části je dále prováděno hledání staticky linkovaného kódu. Knihovní funkce není vhodné ani účelné dekompilovat – zbytečně prodlužují kód a jejich implementace obvykle není pro uživatele dekompilátoru zajímavá. Detekce využívá databázi vzorů známých knihovních funkcí, které jsou převedeny na jednotný formát. Dekódované instrukce jsou nejprve srovnávány se známými vzory a v případě shody jsou nahrazeny voláním odpovídající knihovní funkce.

Přední část nakonec kód analyzuje. Tím se teprve v kódu najdou funkce, globální a lokální proměnné, identifikují se datové typy a eliminuje se mrtvý kód (tj. kód, který není nikdy vykonán).

3.2.3 Střední část (middle end)

Kód vygenerovaný přední částí je sice validní LLVM IR, ale je značně neoptimální. To plyne ze způsobu jeho vzniku – každá instrukce je překládána zvlášť do posloupnosti mikroinstrukcí, které přesně simulují činnost procesoru. Ve výsledku to znamená řádově 10 mikroinstrukcí na operaci, která se často dá vyjádřit instrukcí jedinou, protože vedlejší efekty nejsou pro chod programu důležité. Ze stejného důvodu jsou v kódu jen surové skokové instrukce, nikoli třeba cykly.

Primárním úkolem střední části je tedy kód optimalizovat – odstranit zbytečné, dále nevyužívané mikroinstrukce.

V této fázi jsou také rozpoznány tzv. idiomy. To jsou instrukce či jejich posloupnosti, kterými jsou během překladu programu nahrazovány některé operace za účelem optimalizace chodu programu. Příkladem může být operace xor na stejný registr, která způsobí jeho vynulování rychleji, než kdyby byla nula načítána třeba z paměti.

Střední část vychází z programu opt překladače LLVM. Vstupní kód je procházen v několika průchodech, z nichž každý může provádět nějakou analýzu nebo optimalizaci. Ukázalo se, že řada optimalizací potřebných pro dekompilaci je již v rámci LLVM implementována a infrastruktura LLVM umožňuje vytváření nových průchodů speciálně pro dekompilaci.

3.2.4 Zadní část (back end)

Zadní část překladače má za úkol transformovat optimalizovaný LLVM IR do cílového vysokoúrovňového jazyka. Toho je dosaženo ještě jedním převodem do reprezentace BIR (back-end intermediate representation, vnitřní reprezentace pro zadní část překladače). Jsou detekovány vysokoúrovňové řídící konstrukce, jako jsou podmínky if/else, cykly, příkazy switch a podobně.

Podporovány jsou dva vysokoúrovňové výstupní jazyky, a sice C (norma C99) a takzvaný Python', což je netypovaný jazyk vycházející z Pythonu 3, obohacený o některé konstrukce z C (switch, goto, ukazatele a další).

Stejně jako střední i zadní část staví na projektu LLVM, tentokrát konkrétně na programu llvmir2hll. I v zadní části jsou prováděny další optimalizace, zaměřené na celkovou čitelnost generovaného kódu. Cílový vysokoúrovňový kód je generovaný pomocí průchodu grafy toku řízení (control flow graph).

Kapitola 4

Jazyk C++

C++ je mnohoúčelový objektově orientovaný jazyk (lze v něm však programovat i čistě imperativně). Stírá se zde rozdíl mezi řídícími a datovými konstrukcemi, neboť objekt reprezentuje jak data, tak chování.

V této podkapitole budou popsány základní konstrukce jazyka C++. Tato část čerpá z [27] a [4]. Části, které to vyžadovaly, byly konzultovány s pracovní verzí nadcházejícího standardu C++ [25].

4.1 Charakteristika jazyka

Historie C++ sahá do roku 1979, kdy jeho tvůrce, Bjarne Stroustrup, začal pracovat na rozšíření C o objekty [31]. Pracovně mu tehdy říkal „C with classes“ (C s třídami), protože první verze tohoto jazyka fungovaly jako preprocesor do C, který se pak teprve dál překládal. Již v počátcích byla podporována jednoduchá dědičnost, výchozí hodnoty parametrů funkcí a statický polymorfismus.

Dnešní název C++ dostal jazyk v roce 1983. Tehdy již existoval překladač přímo pro něj (bez mezistupně C), byly implementovány virtuální funkce, mnohonásobná dědičnost, přetěžování funkcí a operátorů a další konstrukce, které ho charakterizují dodnes.

Přes svoji dlouhou historii byl jazyk standardizován až v roce 1998. Zatím poslední přijatá verze standardu je C++11 z roku 2011 [24].

Mezi cíle jazyka podle jeho autora patří [32]:

1. Vlastnosti jsou implementovány do jazyka podle potřeby, nikoli kvůli teoretické čistotě.
2. Programátor, ne jazyk určuje styl programování.
3. Kompatibilita s C.
4. Silný typový systém, který je ale možno explicitně obejít.
5. Zásada „Plať jen za to, co potřebuješ.“: Použití všech prvků jazyka je volitelné a nepoužité vlastnosti nemají žádný dopad na velikost či výkon programu.

C++ v sobě kombinuje vlastnosti nízkoúrovňových i vysokoúrovňových jazyků. Na jednu stranu si ponechává z jazyka C těsný přístup k hardware a programování na nízké úrovni abstrakce, hodí se tedy na psaní ovladačů a firmware vestavěných systémů. Na druhou stranu obsahuje i velmi abstraktní vysokoúrovňové konstrukce, které umožňují pohodlný

a strukturovaný vývoj složitých informačních systémů. Jazyk C++ ovlivnil i mnoho dalších dnes populárních jazyků, z nejznámějších např. Javu a C#. V poslední době je C++ populární i mezi tvůrci virů, proto je vhodné umět i tento jazyk dekompilovat.

4.2 Objekty a třídy

V [17] jsou objekty a jejich třídy v objektově orientovaném jazyce definovány následovně:

„Objekty zapouzdřují jak stav, tak chování. Konkrétně obsahují množinu členských proměnných (také zvaných pole či atributy objektu), které reprezentují stav, a množinu metod, reprezentují chování, které je objekt schopen provést. Metody jsou schopné přistupovat k členským proměnným a měnit jejich hodnoty.“

„Třídy jsou rozšiřitelné ‚šablony,' poskytující výchozí hodnoty polím a těla metodám. Všechny objekty generované ze stejné třídy sdílí stejné metody, ale obsahují samostatné kopie členských proměnných.“

V C++ lze členským proměnným i metodám definovat modifikátor viditelnosti, kterým je možno omezit přístup k prvku na úrovni zdrojových kódů. C++ podporuje 3 typy viditelnosti:

- `private`: K prvku lze přistupovat pouze z metod dané třídy a tříd explicitně označených v této třídě jako přátelské (`friend`);
- `protected`: K prvku lze navíc přistupovat ze tříd, které od této třídy dědí;
- `public`: Přístup k prvku je neomezený.

Kromě tříd existují v C++ i struktury. Struktury obsahující pouze datové atributy se nazývají POD (Plain Old Data, prostá stará data) a jsou v syntaxi i v přeložené binární reprezentaci plně kompatibilní se strukturami v C. Kromě toho ale v C++ mohou i struktury mít metody, dědit od jiných struktur a na svých prvcích specifikovat viditelnost, stejně jako třídy. V takovém případě je mezi strukturami a třídami v C++ jen jeden nepříliš podstatný rozdíl: prvky třídy jsou ve výchozím stavu soukromé (`private`), prvky strutury veřejné (`public`).

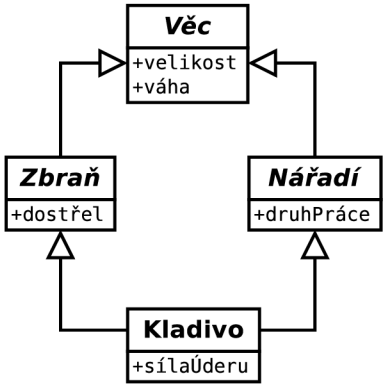
4.3 Metody a operátory

Metody C++ se od běžných funkcí odlišují tím, že jsou definovány v rámci třídy. Při volání každé metody, která není označena klíčovým slovem `static`, se automaticky předává ukazatel na instanci třídy, v jejímž kontextu je metoda volána.

C++ umožňuje přetěžování a přepisování metod. Přetěžování již bylo popsáno: Je možno definovat více metod stejného jména s různým počtem parametrů. Jedná se tedy o více různých metod, které se pouze jmenují stejně.

Přepisování umožňuje v poděděné třídě redefinovat metodu z nadřazené třídy. Tomuto tématu bude věnována část o polymorfismu 4.6.

C++ umožňuje i přetěžování operátorů, tj. jejich definici i pro uživatelsky definované datové typy. Nelze vytvářet operátory nové, pouze definovat sémantiku těm stávajícím. Není však možno měnit aritu operátorů ani prioritu vykonávání.



Obrázek 4.1: Příklad vícenásobného dědění jedné třídy

4.4 Dědičnost

Dědičnost je základním principem objektově orientovaného programování. U třídy můžeme specifikovat jednu či více nadtříd, jejichž atributy a metody budou poděděny i do této třídy. Je tak možno definovat třídní hierarchie.

Na rozdíl od jiných objektových jazyků v C++ neexistuje implicitní kořenová třída, rodič všech ostatních tříd. C++ naopak jako jeden z mála programovacích jazyků podporuje nejen jednoduchou, ale i mnohonásobnou dědičnost, je tedy možné podědit více tříd.

4.5 Virtuální dědičnost

Mnohonásobná dědičnost přináší nové možnosti, ale také problémy. Protože hierarchie tříd je v C++ obecný orientovaný graf a nikoli strom, může se stát, že třída transitivně podědí jinou třídu vícekrát přes různé předky.

To není vždy vhodné. Znamená to, že celá nadřazená třída se všemi svými poli je v poděděné třídě zahrnuta vícekrát. Typický příklad je znázorněn na obrázku 4.1. Vícenásobné zahrnutí nadtřídy má nejen zbytečné paměťové nároky, ale také přináší problémy se synchronizací polí těchto nadtříd. Mají mít vždy stejnou hodnotu? Pokud ano, kdo to zajistí?

Odpovědí C++ na tyto problémy je virtuální dědičnost. Je-li třída poděděna virtuálně, je kompilátorem zajištěno, že všechna pole budou v dědící třídě zahrnuta právě jedenkrát.

Přístup k polím a metodám virtuálně poděděné třídy je ale za běhu programu časově náročnější.

4.6 Polymorfismus, virtuální metody

Polymorfismem (zřídka též česky mnohotvárností) rozumíme v kontextu objektově orientovaného programování volání metody definované v rozhraní (nadtřídě). Implementace této metody se může různit podle konkrétního typu objektu, na němž metodu voláne. Rozhodnutí o konkrétní implementaci může proběhnout už během kompilace (statický polymorfismus), nebo až při běhu programu (dynamický polymorfismus)

Statický polymorfismus je omezenější, ale nepřináší žádné zpomalení. Metoda se vybere během překladu podle deklarovaného typu proměnné objektu. I pokud je do proměnné přiřazen podtyp, který může metodu přepisovat, je použita implementace nadtypu.

Dynamický polymorfismus provádí toto rozlišení až za běhu programu. Implementace metody, zavolaná na jednom konkrétním místě v programu, tak může být rozdílná podle reálně předaného typu objektu. Jelikož rozhodnutí probíhá za běhu, přináší zpomalení. V C++ podle zásady „Plať jen za to, co potřebuješ“ musejí být tyto dynamicky polymorfické metody explicitně označeny klíčových slovem virtual. Říkáme jim tedy virtuální metody.

Zvláštním typem je čistě virtuální metoda (pure virtual method). Takto označená metoda je v nadtřídě pouze deklarována, definice chybí. Třídu obsahující alespoň jednu čistě virtuální metodu nazýváme abstraktní.

Kompilátor nedovolí vytvoření instance abstraktní třídy, její typ však můžeme používat pro uložení objektů s konkrétní implementací.

Pomocí abstraktních tříd, které neobsahují žádné členské proměnné ani jiné metody než čistě virtuální, můžeme napodobit rozhraní (interface), známé z jiných objektových programovacích jazyků.

4.7 Konstruktory a destruktory

Konstruktory a destruktory jsou speciální metody objektu, které slouží pro jeho inicializaci a deinicializaci. Konstruktor může přijímat parametry a může být na jedné třídě přetížen. Destruktor parametry nepřijímá, přetížen být nemůže, ale může být definován jako virtuální.

Jak konstruktor, tak destruktor může mít definované tělo jako jakákoli jiná metoda. Konstruktor navíc inicializuje pole třídy na počáteční hodnoty a volá konstruktory vnořených a poděděných tříd. Destruktor provádí pravý opak.

4.8 Přetypování

Přetypování umožňuje (v jakémkoli jazyce) změnit interpretaci nějaké hodnoty v jednom datovém typu za jiný. Tato konverze může být ztrátová (např. v C desetinné číslo na celé) či jednosměrná.

C++ umožňuje definovat jak implicitní, tak explicitní přetypování z objektu na jiný objekt, z objektu na vestavěný datový typ a naopak.

Využívá se k tomu:

- Konstruktor s jedním argumentem zdrojového typu;
- Operátor přiřazení s argumentem zdrojového typu;
- Operátor přetypování na cílový typ.

Zvláštní pozornost je věnována přetypování ukazatelů. V C je možno explicitně přetypovat libovolný typ ukazatele na libovolný jiný typ ukazatele a změní se pouze logický pohled na data. Adresa reprezentující tento ukazatel zůstane stejná.

Protože typový systém C++ je hierarchický, je nutno rozlišovat podle zdrojového a cílového typu tyto druhy přetypování:

- Upcast – přetypování potomka na předka;
- Downcast – přetypování předka na potomka;
- Obecné přetypování ukazatele jako v C.

Přestože je stále možné používat syntax přetypování jako v C, ze syntaxe není jasné, který druh přetypování máme na mysli. Pokud programátor udělá chybu, snadno místo zamýšleného prvního či druhého případu dostane případ třetí, který skoro jistě způsobí nesprávný chod programu. Jelikož však syntax bude v pořádku, kompilátor nemá možnost chybu odhalit.

C++ proto nabízí 4 nové operátory přetypování:

- `dynamic_cast`: Umožňuje bezpečný upcast i downcast, vyžaduje RTTI;
- `static_cast`: Umožňuje bezpečný upcast, v případě downcastu alespoň kontroluje, že typy jsou příbuzné;
- `reinterpret_cast`: Explicitní vynucení jiného pohledu na data, jsou vyloučeny konverze;
- `const_cast`: Pouze manipulace s modifikátorem const.

4.9 Šablony

Šablony (templates) v C++ umožňují generické programování. Existují ve dvou podobách: šablony funkcí a šablony tříd.

Šablony umožňují popsat algoritmus nezávisle na použitých datových typech. Šablony místo konkrétních typů používají typové parametry. Šablony funkcí se tak často využívají na univerzální popis algoritmů (třídění, řazení apod.), šablony tříd pak například na kontejnery (seznamy, stromy, mapy a další).

Šablony C++ na první pohled připomínají direktivy preprocesoru známé z C (použitelné i v C++). Na rozdíl od nich ale v šablonách lze využít znalost o typovém systému a překladač je může lépe optimalizovat.

Samotná definice šablony nezpůsobí generování žádného strojového kódu. To nastane až při instanciaci šablony, tedy použití s konkrétními typovými parametry. Pro každou kombinaci typových parametrů pak dojde k vygenerování jedné kompletní instance šablony.

4.10 Výjimky

Výjimky v programovacích jazycích slouží k vyřešení chybových situací při zachování rozumné čitelnosti kódu. Není tedy třeba testovat chybový stav po každé operaci.

Podpora výjimek je nepovinná a lze ji při kompilaci vypnout.

Blok kódu může být označen pomocí klíčových slov `try` a `catch`, výjimka je pak vyvolána pomocí `throw`. Vyvolání výjimky způsobí přerušení toku programu a vyvolání nejbližšího ošetření výjimky v bloku `catch` (exception handler). Výjimka může být ošetřena na jiné úrovni kódu, než kde byla vyvolána, proto je zajištěno automatické uvolnění všech objektů, které se ocitly mimo rozsah platnosti.

Bloků `catch` může být definováno více, každý z nich může specifikovat typ výjimky, kterou ošetřuje. Tento typ odpovídá objektu předanému příkazu `throw`. Je povolena konstrukce `catch (...)`, která odchytává vše.

Pokud není výjimka odchycena ani ve funkci main, způsobí ukončení programu s chybovým hlášením.

Kapitola 5

Dekompilace C++

Jazyk C++ je v porovnání například s C jazyk vysokoúrovňový, poskytující vývojářům již poměrně vysokou míru abstrakce, a proto se zdrojový kód již značně liší od strojového kódu, který vznikne po přeložení. I přes to jsou podporovány i některé nízkoúrovňové operace, umožňující například ničím nekontrolovatelné přetypování (`reinterpret_cast`).

Tyto vlastnosti dělají z dekompilace jazyka C++ obtížný úkol, který je z principu vždy nedokonalý – některé informace jsou při překladu jednoduše ztraceny, například modifikátory viditelnosti.

Tato kapitola vychází z článků [21, 20, 30, 22], ze standardu [25] a z vlastního experimentování se zpětným překladačem AVG.

5.1 Konstrukce jazyka

Standard C++ specifikuje pouze API (Application Programming Interface, rozhraní na úrovní zdrojového kódu), ale nediktuje konkrétní implementaci ani ABI (Application Binary Interface, binární rozhraní).

Přestože kompilátorů C++ a platforem existuje velké množství, jsou techniky použité pro implementaci konstrukcí jazyka podobné.

V praxi rozlišujeme dva hlavní proudy C++ překladačů:

1. Microsoft Visual C++
2. GCC a Clang

V následujícím textu tedy budu brát v úvahu oba proudy. Pokud není druh překladače uveden, dané tvrzení platí univerzálně ve všech podporovaných překladačích.

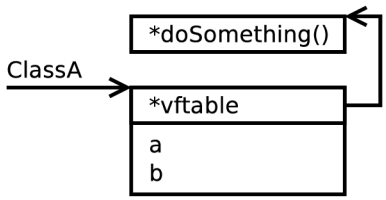
5.1.1 Členské funkce (metody)

Členské metody definované jako statické se příliš neliší od obyčejných funkcí. Jsou omezené pouze lexikálně jménem daného objektu, případně viditelností, ale z pohledu dekompilace je lze překládat jako jakékoli jiné funkce jazyka C.

Nestatické metody se vždy volají na nějakém objektu, je tedy třeba funkci předat ukazatel na tento objekt (`this` pointer). Ve většině překladačů a architektur se tak děje pomocí implicitního prvního parametru, který se předává stejně jako explicitní parametry.

Výjimku tvoří Visual C++ na architektuře x86 a Windows, který využívá speciální volací konvenci `__thiscall` [14] a ukazatel na `this` předává vždy v registru ECX. Stejnou

```
class ClassA {
    public:
        int a, b;
        virtual void doSomething();
};
```



Obrázek 5.1: Třída ClassA a její rozložení v paměti

konvenci používá i překladač GCC na Windows a x86, pravděpodobně z důvodů binární kompatibility s Visual C++. Na ostatních překladačích a platformách (mj. i GCC pro Linux) se nevirtuální členské funkce pro dekompilaci nijak neliší od nečlenských funkcí, které v prvním parametru explicitně dostávají ukazatel na instanci třídy.

Spolehlivě odlišit členské a nečlenské funkce proto v obecném případě není možné a toto rozlišení není součástí mojí práce.

5.1.2 Virtuální metody

Všechny podporované překladače implementují virtuální funkce pomocí tabulek virtuálních funkcí (vftable) v paměti.

Mějme třídu ClassA. Jakmile má třída byť jen jedinou virtuální metodu (ať již vlastní nebo poděděnou), musí pro ni existovat tabulka virtuálních funkcí. Jedná se o seznam ukazatelů na funkce, implementace virtuálních metod třídy. Rozložení třídy ClassA v paměti je znázorněno na obrázku 5.1

Třída samotná je pak rozšířena o ukazatel na tuto tabulku, který je naplněn v konstruktoru (a též destruktoru).

Při volání virtuální metody využíváme faktu, že ukazatel na vftable je první položkou struktury. Samotná funkce pak přejímá ukazatel na svůj objekt pomocí nultého skrytého parametru. Za zmínku stojí, že k takto komplikovanému volání dojde pouze v případě, že používáme ukazatel na ClassA (či referenci, která je vnitřně implementovaná pomocí ukazatele). Pokud bychom používali přímo instanci ClassA, bude vždy zavolána přímo implementace dané metody bez použití vftable, jelikož už v době kompilace je jasné, která implementace musí být použita.

Stejná metoda může tedy někdy být volána přímo a jindy pozdní vazbou pomocí vftable.

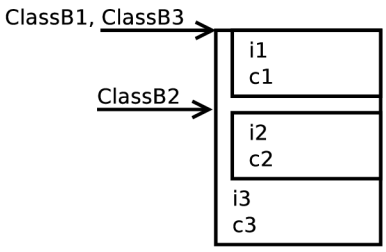
5.1.3 Dědičnost

Jednoduchou i mnohonásobnou nevirtuální dědičnost tříd, které nemají virtuální funkce, můžeme vyjádřit pomocí kompozice.

Třída ClassB3 je uložena v paměti, jak ukazuje obrázek 5.2.

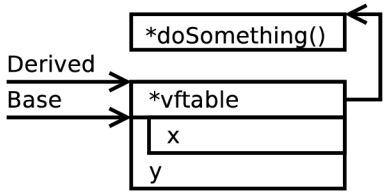
Nadřazené třídy jsou vloženy na začátek poděděné třídy. Typicky jsou sešazeny ve stejném pořadí, v jakém jsou poděděny, v dalším textu ale uvidíme, že to není vždy pravidlem. Kompilátory mohou mezi jednotlivé poděděné struktury vložit několik nevyužitých bytů (padding) kvůli zarovnání v paměti – toto ostatně platí pro položky jakékoli struktury v C i C++. Jak je patrné z obrázku, offset každé položky, poděděné i přímé, je konstantní, proto se přístupy přeloží do strojového kódu identicky.

```
class ClassB1 {
    public:
        int i1; char c1;
};
class ClassB2 {
    public:
        int i2; char c2;
};
class ClassB3 : public ClassB1, public ClassB2{
    public:
        int i3; char c3;
};
```



Obrázek 5.2: Třída ClassB3 a její rozložení v paměti

```
class Base {
    public:
        int x;
};
class Derived {
    public:
        int y;
        virtual void doSomething();
};
```



Obrázek 5.3: Rozložení v paměti pro třídu Derived s virtuální tabulkou, která dědí od třídy Base bez virtuální tabulky

5.1.4 Přetypování

Přetypování na nadřazenou třídu odpovídá v C vypočítání počáteční adresy vnořené struktury, tedy vlastně opět přístupu k položce struktury. Je pouze nutné počítat s případem, kdy přetypovaný ukazatel je NULL. Tato hodnota musí být zachována i po přetypování.

Pro ukázku využívám třídu ClassB3.

```
// C++
ClassB3 *b = new ClassB3();
ClassB1 *b1 = b;
ClassB2 *b2 = b;
```

```
// C
struct ClassB3 *b = new_ClassB3();
struct ClassB1 *b1 = (struct ClassB1 *) b;
struct ClassB2 *b2 = b ? &b->b2 : NULL;
```

Zvláštním případem je první poděděná struktura. Ta vždy začíná na počáteční adrese struktury potomka, tedy na offsetu 0. Toto přetypování tedy není vůbec třeba překládat do strojového kódu, ani není není třeba kontrolovat NULL.

5.1.5 Virtuální tabulky a jednoduchá dědičnost

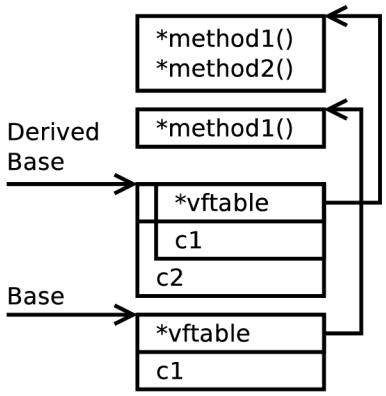
Ve všech zkoumaných překladačích je zvykem, že ukazatel na tabulku virtuálních funkcí je první položkou třídy (na offsetu 0). Dědíme-li od třídy, která nemá žádné virtuální funkce, ale dědící třída nějaké definuje, pak uložení takové třídy bude vypadat jako na obrázku 5.3.

Nejprve je ukazatel na vftable, teprve poté obsah poděděné třídy, za kterým následují vlastní pole dědící třídy.

Pokud se dědí od třídy, která již virtuální tabulku má, tato se pouze rozšíří o případné nově definované virtuální funkce, které se připíší na konec. Nevytváří se tedy nové skryté pole ve struktuře, nýbrž využije se ukazatel z poděděné třídy, který je příhodně umístěn na offsetu 0 celé třídy.

Rozložení tříd je ilustrováno na obrázku 5.4.

```
class BaseClassC {
    public:
        virtual void method1();
        char c1;
};
class DerivedClassC: public BaseClassC {
    public:
        virtual void method1();
        virtual void method2();
        char c2;
};
```



Obrázek 5.4: Rozložení třídy DerivedClassC v paměti

Při volání je teď nutno rozlišovat, na kterém typu byla virtuální funkce poprvé definována, a podle toho použít správnou virtuální tabulku.

5.1.6 Virtuální tabulky a vícenásobná dědičnost

Nejprve vyřeším situaci, kdy třída sice dědí od více tříd, ale jen jedna z těchto nadtříd definuje virtuální funkce. V takovém případě překladač přeuspořádá nadtřídy tak, aby ta s virtuální tabulkou byla na prvním místě, a sloučí virtuální tabulky tak, jak bylo popsáno v předchozí kapitole. Ostatní nadtřídy pak vyřeší kompozicí.

Pokud má třída dědit od více nadtříd s více virtuálními tabulkami, je první z nich upravena již popsaným způsobem. Další nadtřídy jsou zakomponovány do podtřídy včetně ukazatelů na své virtuální tabulky. Výsledná třída má tedy více tabulek virtuálních funkcí.

Paměťové rozložení je na obrázku 5.5.

U těchto dalších virtuálních tabulek je nutné dát pozor na to, jaký typ se předává jako ukazatel na `this`. V kapitole o přetypování bylo již ukázáno, že v případě mnohonásobné dědičnosti ukazatele na nadtřídu a podtřídu neukazují na stejnou adresu.

Metody nadtřídy očekávají jako this ukazatel na sebe. Aby fungoval polymorfismus, musí být zajistěno, aby i v případě přepsaných metod bylo možné tyto metody volat s ukazatelem na nadtřídu, zároveň ale v těle funkce musí být možné přistupovat k atributům podtřídy.

Zde se implementace v jednotlivých překladačích liší. GCC a Clang vytvoří nový záznam v primární vftable a do tabulek dalších předků uloží ukazatel na tzv. adaptérovou metodu. Jedná se kód automaticky generovaný překladačem, který provede přetypování ukazatele na `this` a zavolá cílovou metodu.

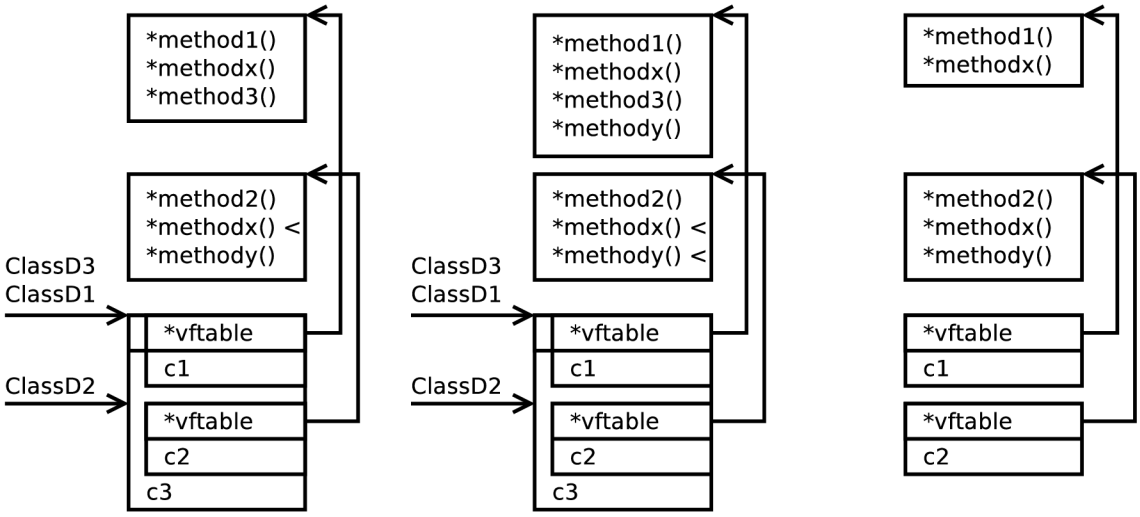
Visual C++ potřebu zvláštních adaptérů částečně eliminuje tím, že přetypování vloží přímo na začátek implementace cílové metody. V případě, že je přetížena metoda z více předků, je již samostatná adaptérová metoda nezbytná vždy.

5.2 Metody dekompilace

V této kapitole popíši několik způsobů, jak dekompilovat vybrané konstrukce jazyka C++. Zaměřuji se na získání co nejpřesnější rekonstrukce datového modelu.

```
class ClassD3: public ClassD1, public ClassD2 {
    public:
        virtual void method3();
        virtual void methodx();
        virtual void methody();
        char c3;
};
```

```
class ClassD1 {
    public:
        virtual void method1();
        virtual void methodx();
        char c1;
};
class ClassD2 {
    public:
        virtual void method2();
        virtual void methodx();
        virtual void methody();
        char c2;
};
```



Obrázek 5.5: Třída ClassD3 a její rozložení pro jednotlivé překladače: Visual C++ (vlevo), GCC a Clang (uprostřed). Pro srovnání i rozložení samostatných tříd, od kterých dědí (vpravo). Adaptérové metody jsou označeny šipkou.

5.2.1 Detekce virtuálních tabulek

Nalezení záznamů ve virtuálních tabulkách [21], uložených v datových sekcích programu, je prvním krokem dalších analýz.

Záznamy v tabulce virtuálních funkcí (vftable) jsou ukazatele na funkce, vftable je pak posloupnost takovýchto ukazatelů, na jejíž první záznam je odkazováno z kódu (v konstruktorech a destruktorech) a na další záznamy ukazováno není. Ukazatel na funkci rozeznám jako slovo o bitové délce odpovídající dekompilované architektuře, jehož hodnota je adresou do kódového segmentu.

Tímto způsobem získám dostatečnou představu o tabulkách virtuálních funkcí, které stačí jako vstup dalším analýzám.

5.2.2 Analýza RTTI

RTTI (Run-Time Type Information) poskytuje informace o typech tříd, struktur a dalších datových typech, jejich názvu, dědičné hierarchii a jejich metodách [29]. Tyto informace jsou použity u operátoru dynamic_cast a typeid, který vrátí textovou reprezentaci typu předaného objektu.

Struktura záznamů RTTI je dána ABI (Application Binary Interface, binárním rozhraním) překladače a samotné záznamy jsou uloženy v datovém segmentu, stejně jako virtuální tabulky. Ukazatel na RTTI je typicky uložen před začátkem vftable dané třídy. Pro úspěšnou dekompilaci proto stačí detekovat pozice virtuálních tabulek a znát ABI daného překladače.

Využití RTTI pro univerzální dekompilaci bohužel není možné. RTTI je nepovinnou součástí jazyka a lze ho vypnout. Mnoho programátorů RTTI nepoužívá, dokonce i první autor jazyka Bjarne Stroustoup se nejprve obával, že RTTI bude často zneužíváno [31]. Další se bojí výkonnostního propadu či vyšší paměťové náročnosti. Některé projekty se také snaží dynamické přetypování vyřešit po svém, např. Qt[1] nebo LLVM[2].

RTTI je nepovinné a jeho přítomnost dekompilaci velmi zjednodušuje. Toho jsou si vědomi i programátoři, kteří se snaží dekompilaci svých programů zabránit. Především autoři virů dělají vše pro to, aby byli vůči dekompilaci imunní, proto nelze očekávat, že by dekompilátorům takto ulehčovali práci.

5.2.3 Analýza virtuálních tabulek

Tento postup určuje hierarchii tříd [21] podle hierarchie tabulek virtuálních funkcí (vftable), které k nim přísluší. Může se stát, že k jedné vftable přísluší i více tříd v příbuzenském poměru (pokud potomek nedefinuje žádné virtuální metody, je jeho vftable stejná jako předka a může být použita znovu).

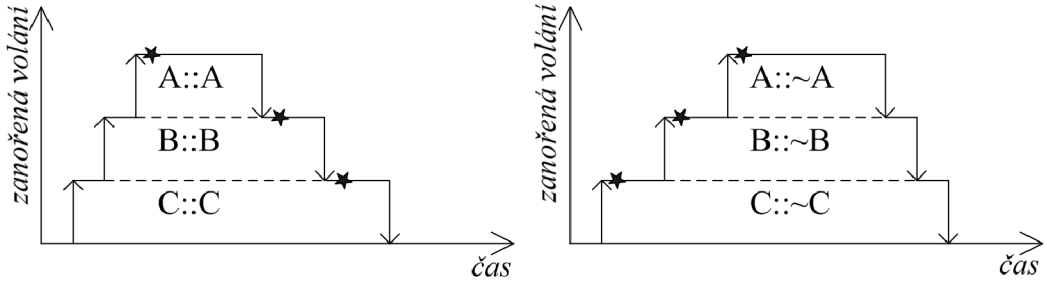
Na rozdíl od tříd, tabulky podporují pouze jednoduchou dědičnost. (Mnohonásobná dědičnost tříd je implementována pomocí více tabulek).

Mám-li vftables B (Base) a D (Derived) je definována relace přímé báze tak, že jedna z tříd odpovídající B je přímou bází jedné ze tříd odpovídající D. Relace jednoduché dědičnosti mezi tabulkami je pak transitivním uzávěrem relace přímé báze.

Analýza pak probíhá pomocí následujících pravidel:

1. Pokud tabulka B má méně prvků než tabulka D, pak B nedědí od D (v C++ nelze odstranit zděděné virtuální metody)

[1] http://qt-project.org/
[2] http://llvm.org/



Obrázek 5.6: Vnořené volání konstruktorů a destruktorů. Hvězdičky označují inicalizaci ukazatelů na virtuální tabulky a případně následné provádění těla konstruktoru (převzato z [20])

2. Pokud metoda B na indexu i (značíme B_i) je čistě virtuální (pure virtual) a D_i není, pak B nedědí od D (nelze nahradit konkrétní implementaci abstraktní metodou)

3. Pokud pro nějakou metodu i není stejná deklarace v B_i a D_i, B a D nejsou v dědičném poměru (lze přepisovat jen metody se stejnou deklarací)

Tímto způsobem získám množinu omezení mezi virtuálními tabulkami a potažmo i odpovídajícími třídami. Je pravděpodobné, že z daných omezení nevyplyne jediné řešení stromu dědičnosti virtuálních tabulek. Pro jeho získání jsou potřeba další analýzy.

5.2.4 Analýza konstruktorů a destruktorů

Abych získal konkrétní relace dědičnosti, je nutno detekovat a analyzovat takzvané speciální funkce: konstruktory a destruktory.

Jejich společnou vlastností je, že inicializují ukazatel na virtuální tabulku (či více tabulek) a v rámci svého provádění volají konstruktory, resp. destruktory nadřazených tříd.

Konstruktor:

1. Zavolá konstruktory nadřazených tříd.

2. Zavolá konstruktory členských tříd.

3. Inicializuje ukazatel (ukazatele) na tabulku (tabulky) virtuálních funkcí a provede uživatelsky definované tělo konstruktoru.

Destruktor provádí akce inverzní ke konstruktoru:

1. Inicializuje ukazatel (ukazatele) na tabulku (tabulky) virtuálních funkcí a provede uživatelsky definované tělo destruktoru.

2. Zavolá destruktory členských tříd v opačném pořadí.

3. Zavolá destruktory nadřazených tříd v opačném pořadí.

Pro strom tříd A, B, C, kde B dědí od A a C dědí od B, je provádění speciálních funkcí znázorněno na obrázku 5.6.

Konstruktory a destruktor tříd obsahující virtuální funkce lze od ostatních funkcí rozeznat primárně podle inicializace virtuálních tabulek (potencionální virtuální tabulky máme již detekované pomocí již popsané analýzy). Poté lze sledovat jejich strukturu, a tím rozlišit

konstruktor, destruktor a případně jinou funkci, která náhodou přistupuje k ukazateli na funkci.

Za zmínku stojí, že vnořená volání konstruktorů (resp. destruktorů) mohou být inlinovaná – kód volané funkce je překladačem vložen přímo do těla volající funkce za účelem rychlejšího provádění. Opakované inicializace jednoho paměťového místa na adresy různých virtuálních tabulek mohou být pak překladačem v rámci optimalizací odstraněny, čímž se analýza značně ztíží.

Kapitola 6

Návrh řešení

V této kapitole je popsáno řešení dekompilace C++ v rámci dekompileru AVG. Metoda používá mnoho postupů, které již byly popsány v předchozích kapitolách.

6.1 Cíle dekompilace

Tato práce si klade za cíl analyzovat a identifikovat ze strojového kódu programu následující konstrukce jazyka C++:

1. Třídy, třídní hierarchie
2. Konstruktory a destruktory
3. Virtuální funkce
4. Volání virtuálních funkcí

Vzhledem k nepatrnému rozdílu mezi třídou a strukturou v C++ budu v následujícím textu pod pojmem *třída* myslet třídu či strukturu s virtuálními metodami a pod pojmem *struktura* třídu či strukturu bez virtuálních metod. Tato práce se zabývá pouze třídami (s virtuálními metodami).

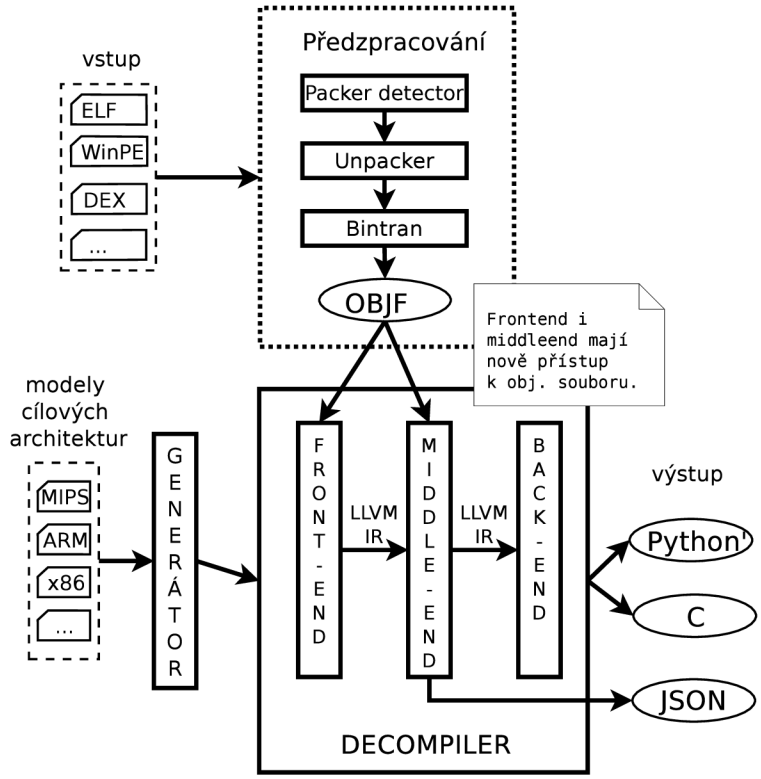
Struktury C++ se v principu nijak neliší od struktur C (dědičnost je nahrazena kompozicí) a jsou již detekovány jinými analýzami na úrovni přední části překladače.

6.2 Architektura

Nosná část mojí práce je implementována ve střední části překladače (middle end). Architektura LLVM a především jazyk vnitřní reprezentace LLVM IR se ukázaly jako velmi vhodné nástroje pro vysokoúrovňové analýzy a modifikace, jako je detekce C++. Menší zásahy musí být provedeny i v přední části překladače.

Bylo však nutné mírně změnit celkové schéma částí překladače AVG, jak je znázorněno na obrázku 6.1. Nejen přední, ale i střední části překladače byl umožněn přístup ke vstupnímu datovému souboru, protože některé informace nutné pro analýzy C++ (například tabulky virtuálních funkcí) nebyly součástí LLVM IR generovaného přední částí překladače.

Tato práce je zaměřena na analýzu a detekci konstrukcí jazyka C++. Výstupní kód tedy je stále C či Python a tato práce ho mění pouze v nezbytné možné míře a vždy v rámci existující syntaxe. Úpravy zadní části proto nejsou součástí této práce.



Obrázek 6.1: Upravené schéma dekompilátoru AVG

Zatímco primárním výstupem pro lidského uživatele dekompilátoru zůstává kód generovaný backendem, pro ukládání strukturovaných dat není vhodný. Stejně tak nevyhovují ani metadata LLVM IR, která se těžko zpracovávají externími skripty a vizuálně jsou nepřehledná. Dalším výstupem práce proto je nově definovaný deskriptorový soubor ve formátu JSON [8], do něhož jsou ukládány výsledky všech analýz hierarchie pro další strojové zpracování. Tento soubor má využití v automatických testech (viz kapitola 8), ale v budoucnu i v dalších nástrojích dále zpracujících výsledky dekompilace (doplněk do disassembleru IDA, vizualizace třídní hierarchie, atd.)

6.3 Úpravy přední části

V přední části překladače je především nutno zajistit, aby byly detekovány a přeloženy opravdu všechny potřebné funkce.

Součástí činnosti frontendu je analýza detekovaného sledu instrukcí a jejich seskupování do základních bloků a funkcí. Dnešní překladače však velmi často generují posloupnosti instrukcí, které ve skutečnosti funkcemi nejsou. Jedná se o zaváděcí kód, v případě C++ rutiny pro obsluhu výjimek a podobně. Tento kód není pro dekompilaci důležitý a jeho další zpracování zbytečně prodlužuje dobu dekompilace. Proto pokud přední část dokáže detekovat hlavní funkci (`main`), jsou všechny funkce nedosažitelné z hlavní funkce odstraněny.

Problémem jsou virtuální funkce jazyka C++, které většinou přímo volány nejsou. Stejný problém nastává u adaptérového kódu (anglicky code thunks), který je používán při volání virtuálních funkcí.

Je tedy třeba již ve frontendu provést základní analýzu datové sekce, v níž jsou hledány ukazatele na funkce, které mohou tvořit tabulky virtuálních funkcí. Takto odkazované funkce pak musejí být označeny jako dosažitelné z funkce `main`, a tím chráněné proti odstranění.

Vzhledem k tomu, že výsledky této analýzy jsou použity pouze pro tyto účely, není třeba provádět sofistikovanou detekci tabulek virtuálních funkcí, jako byla popsána v kapitole 5.2.1, ale stačí nalezení ukazatelů na funkce v datových sekcích. Prochází se proto datové sekce po jednotlivých adresách (krok odpovídá bitové délce dané architektury), a pokud hodnota na dané adrese odpovídá adrese počátku existující funkce, je tato funkce označena jako dosažitelná.

Tímto způsobem lze detekovat všechny virtuální funkce a dopad špatné detekce je minimállní – pouze se některá zbytečná funkce neodstraní.

6.4 Střední část

Analýzy dekompilovaného jazyka C++ i optimalizace kódu vnitřní reprezentace plně využívají možností platformy LLVM a jsou implementovány jako průchody částmi LLVM IR, které na sobě vzájemně závisí. Klíčové analýzy jsou popsány v této podkapitole.

6.4.1 Hledání tabulek virtuálních funkcí

Analýza provádí detekci ukazatelů na funkce podle metody popsané v kapitole 5.2.3. Jako jediný průchod ve střední části překladače přistupuje nejen k LLVM IR vytvořenému ve front endu, ale také k surovému objektovému souboru. Je to dáno tím, že v přední části není možné rozhodnout, které části datové sekce obsahují užitečná data.

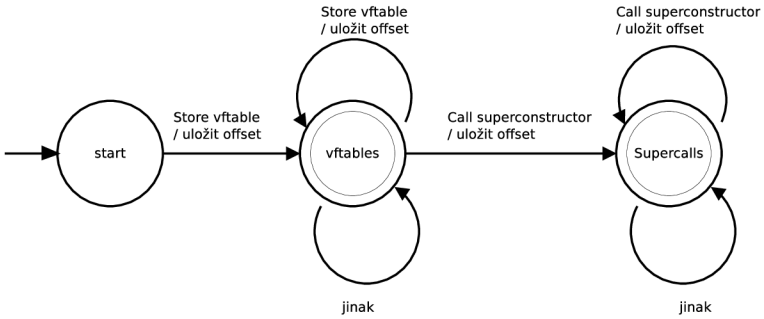
Z objektového souboru lze vyčíst adresové rozsahy datových sekcí a tím detekovat ukazatele na funkce. První část analýzy pak probíhá následovně:

1. Z objektového souboru jsou načteny informace o datových sekcích.
2. Postupně jsou slovo po slovu procházeny datové sekce a pro každou hodnotu je rozhodnuto, zda se jedná o potencionální ukazatel na funkci – adresu spadající do kódové sekce, na které začíná detekovaná funkce.
3. Výsledky jsou uloženy pro zpracování dalšími průchody.

Protože LLVM IR pochájející z front endu nemá žádnou informaci o virtuálních tabulkách, je inicializace ukazatelů na ně dekompilována jako přiřazení konstanty – adresy do datové sekce – do registru či paměťového místa. V druhé části analýzy jsou vyhledávána tato přiřazení, a pokud adresa daná konstantou skutečně obsahuje dříve detekovaný ukazatel na funkci, je tato adresa považována za počátek tabulky virtuálních funkcí.

Pro kompletní vftable musí platit tato omezení:

1. Její první prvek je referencován.
2. Její další prvky referencovány nejsou.
3. Obsahuje pouze ukazatele na funkce – oproti algoritmu implementovanému ve frontendu zde již nestačí, že adresa ukazuje do kódové sekce, ale na dané adrese musí začínat dekompilovaná funkce.



Obrázek 6.2: Detekce konstruktorů a destruktorů

Průchod tedy zkoumá přiřazování konstant, které by mohly být adresy do datové sekce. Je-li na dané adrese ukazatel na funkci, je uložena dvojice (A_{start}, Is), kde A_{start} je adresa počátku vftable a Is je seznam instrukcí přistupující na tuto adresu. Takto je prozkoumán celý dekompilovaný kód.

Poté je ke každé dvojici zjištěna maximální adresa A_{end} taková, že $\forall A_i \ : \ A_{start} \leq A_i \leq A_{end} \ : \ fnptr(A_i) \ \wedge \ A_i \neq A^x_{start}$, kde $fnptr(...)$ označuje ukazatel na funkci a A^x_{start} je počátek jakékoli jiné tabulky. Výsledkem je tedy seznam trojic (A_{start}, A_{end}, Is).

Posledním stupněm je úprava kódu LLVM IR, aby reflektoval získané znalosti. Pro každou trojici (A_{start}, A_{end}, Is) je vytvořena struktura s unikátním typem. Jejími položkami jsou typované ukazatele na funkce, jejichž definice je zjištěna podle konkrétní funkce LLVM IR nalezené podle adresy. Pro každou strukturu je definována globální instance, kde jsou ukazatelům přiřazeny implementace.

Nakonec jsou všechny instrukce ze seznamů Is nahrazeny instrukcemi, které přiřazují adresu globální instance tabulky namísto číselné konstanty.

V LLVM je tabulka virtuálních funkcí reprezentována takto:

```
%vftable_4020e0_type = type { void ()* }
@vftable_4020e0 = global %vftable_4020e0_type { void ()* @function_401040 }
```

Typ i instance jsou pojmenovány podle počáteční adresy tabulky.

6.4.2 Detekce konstruktorů a destruktorů

Zde je prováděna klasifikace funkcí podle postupu popsaného v kapitole 5.2.4. Funkce obsahující instrukce ze seznamů Is z předchozích průchodů jsou kandidáty na konstruktory či destruktory. Tato analýza má za cíl rozlišit konstruktor od destruktoru a vyfiltroval funkce, které speciálními funkcemi vůbec nejsou.

Je využíváno faktu, že ačkoli jak konstruktory, tak destruktory mohou obsahovat jakýkoli uživatelsky definovaný kód, jejich začátek (v případě konstruktorů a destruktorů) či konec (jen u destruktorů) obsahuje speciální instrukce generované překladačem. Tyto instrukce navíc neobsahují skoky, takže stačí analyzovat úvodní a koncový základní blok funkcí.

Analýza je definována pomocí stavového automatu 6.2. Na začátku konstruktoru a destruktoru jsou očekávány instrukce přiřazení virtuálních tabulek do lokálních proměnných, každý konstruktor i destruktor musí obsahovat alespoň jednu (struktury bez virtuálních funkcí neanalyzujeme, viz 6.1). V nasledujícím stavu jsou zkoumána volání funkcí, které jsou také na seznamu Is – to jsou pravděpodobná volání konstrutorů a destruktorů nadřazených tříd. Ty hledáme na dvou místech: V úvodním bloku hned po virtuáních tabulkách a

na konci koncového bloku. Jejich přítomnost na jednom z těchto umístění rozhoduje o tom, zda se jedná o konstruktor, nebo destruktor.

Navíc jsou v obou stavech zkoumány offsety, do kterých jsou ukládány adresy tabulek, respektive na kterých jsou volány nadřazené funkce. Takto získáme informace o paměťovém rozložení nadtříd a tabulkách ve třídách.

V případě, že se nepodaří nalézt žádné nadřazené metody, nelze rozhodnout, zda se jedná o konstruktor či destruktor. Pokud je nalezena alespoň tabulka, je funkce zaevidována pro oba případy. V opačném případě se s největší pravděpodobností nejedná o speciální funkci a z dalších analýz je vypuštěna.

Výsledkem analýzy je seznam detekovaných speciáních funkcí spolu s příznaky, zda se jedná o konstruktor či destruktor, a seznamem referencovaných tabulek včetně offsetů a seznam volaných nadřazených funkcí, taktéž včetně offsetů.

6.4.3 Analýza třídní hierarchie

Třídní hierarchie je rekonstruována z výsledků přechozích analýz. Analýza konstruktorů a destruktorů detekovala speciální funkce a pro každou zjistila referencované virtuální tabulky. Speciální funkce jsou seskupeny podle tohoto seznamu – předpokládáme, že kombinace tabulek je pro každou třídu unikátní. Tento předpoklad však neplatí pro seznam metod nadřazených tříd, protože konstruktorů může být na jedné nadtřídě definováno více.

Poté, co jsou rozpoznány třídy, jsou k nim přiřazeny již známé speciální metody, virtuální tabulky a na základě volání jejich metod také nadtřídy.

Na tomto místě je možné zkontrolovat, zda tabulky splňují omezení z 5.2.3. Získané informace jsou převedeny do výstupního formátu a uloženy.

6.4.4 Analýza použití globálních proměnných

Pro činnost následujících analýz je nutné vědět, jak se které funkce chovají ke globálním proměnným (v LLVM IR to zahrnuje i registry). Především zajímavé jsou následující situace:

1. Čtení neinicializované globální proměnné,
2. Jakýkoli zápis do globální proměnné.

Analýza je podobná běžně využívané analýze Reaching Definitions [1], kdy program procházíme po jednotlivých základních blocích podle CGF (Control Flow Graph, graf toku řízení) a v nich po instrukcích ve směru vykonávání programu. Průběžně udržujeme dvě množiny pro každý základní blok: L_{read} a L_{write}, na začátku jsou oba prázdné. Algoritmus pro základní blok BB je znázorněn na výpise 6.3:

1. Pro každý základní blok BB zjistíme příspěvek z bloků předcházejících. Pokud některý z nich zatím není zpracovaný, počítáme prozatím s prázdnou množinou;
2. Instrukce BB procházíme proti směru vykonávání;
3. Narazíme-li na zápis do globální proměnné G, přidáme G do množiny L_{write}. Zároveň musíme G odebrat z množiny L_{read}, protože jakékoli čtení muselo následovat v programu za tímto zápisem, a proto se nemohlo jednat o čtení neinicializované hodnoty.
4. Narazíme-li na čtení z G, optimisticky předpokládáme, že je G neinicializovaná, a proto ji přidáme do množiny L_{read}.

1. $L^{BB}_{write} \leftarrow \bigcup_{P \in pred(BB)} L^{P}_{write}, \quad L^{BB}_{read} \leftarrow \bigcup_{P \in pred(BB)} L^{P}_{read}$

2. $Forall\ I \in instructions(BB)\ in\ reverse\ order:$

3. $If\ I = Store(G): L^{BB}_{write} \leftarrow L^{BB}_{write} \cup \{G\}, \quad L^{BB}_{read} \leftarrow L^{BB}_{read} - \{G\}$

4. $If\ I = Load(G): L^{BB}_{write} \leftarrow L^{BB}_{write} \cup \{G\}$

5. $If\ I = Call(f): L^{BB}_{write} \leftarrow L^{BB}_{write} \cup L^{f}_{write}, \quad L^{BB}_{read} \leftarrow L^{BB}_{read} - L^{f}_{write}, \quad L^{BB}_{read} \leftarrow L^{BB}_{read} \cup L^{f}_{read}$

Obrázek 6.3: Použití globálních proměnných

5. Volání funkce způsobí přidání všech proměnných z množin funkce L^{f}_{write} a L^{f}_{read} do odpovídajících množin BB a odebrání proměnných, do kterých bylo zapisováno, z množiny L_{read}.

Obsahuje-li CFG cyklické závislosti, jsou bloky v těchto cyklech evaluovány opakovaně, dokud algoritmus nekonverguje k pevnému bodu. Množiny L^{f}_{read} a L^{f}_{write} pro funkci jsou převzaty z jejího vstupního bloku.

Zvláštní péče je věnována případu, kdy pro nějakou proměnnou G a funkci f platí, že $G \in L^{f}_{read}$ a zároveň $G \in L^{f}_{write}$. V takovém případě jsou zkoumány první čtení a poslední zápisy a zjišťuje se, zda není vždy v posledním zápise navrácena původní hodnota proměnné, uložené po celou dobu trvání funkce v lokální proměnné. Je-li tento případ nalezen, je G odstraněna z množiny L_{write}.

6.4.5 Detekce a propagace datových typů

Typový systém C++ je složitější než C a prvotní rozpoznání jednoduchých i složených datových typů v dekompilátoru AVG probíhá v přední části, která nemá žádné informace o třídách a virtuálních tabulkách.

Základem rozpoznávání je analýza datových toků (data-flow). Algoritmus musí pracovat s těmito omezujícími podmínkami:

1. Narozdíl od typů v LLVM IR, přiřazený typ C++ pro jednu danou proměnnou může být na různých místech programu rozdílný.

2. Uvažujeme-li typ ukazatel, může docházet k situaci, že dvě rozdílné proměnné ukazují na stejné místo v paměti (pointer aliasing), a měly by tak mít stejný typ.

3. Typy globálních proměnných (tj. i registrů) .mohou být změněny ve volaných funkcích.

Typickým příkladem prvního omezení je volací konvence `__thiscall`, kdy registr ECX obsahuje vždy ukazatel na třídu takového typu, který je potřeba v právě volané funkci.

Řešením je ukládat typovou informaci vždy vzhledem k běhu programu, tedy vztaženou ke konkrétní instrukci.

Druhé omezení je řešeno nepřímo pomocí propagačního algoritmu. Řešení třetího bodu využívá přechozí analýzy přístupu ke globálním datům. Princip analýzy je následující:

1. Výchozím bodem je volání konstruktorů. Ty definují typ svého prvního parametru, svůj návratový typ a také typ registru ECX (`__thiscall`) v době jejich volání.
2. Pro každou hodnotu, jejíž typ je nalezen, je tento typ rekurzivně propagován směrem k definici (vzhůru) i ke všem použitím (dolů).
3. Mnoho instrukcí má zvláštní pravidla, která musejí platit pro jejich operandy a návratovou hodnotu.
   (a) Instrukce přetypování (LLVM typů) C++ typ nemění.
   (b) Pro ukládání (`Store`) a načítání (`Load`) platí, že paměťový operand je typu ukazatel na hodnotový operand.
   (c) Přičítání (`Add`) konstanty k ukazateli nebo instrukce `GetElementPtr` mění offset, který je zpracováván v následujícím načítání.
   (d) a další.

Při propagaci typů napříc základními bloky a funkcemi využívám znalosti o offsetech nadtříd. Ukazatel, jehož typ je byl určen na jistou třídu, je zároveň ukazatelem na všechny přímé i nepřímé předky této třídy, které v této třídě začínají na nulovém offsetu. Na ostatní předky, které leží na nenulovém offsetu, by musel ukazatel být v dekompilovaném kódu explicitně přetypován přičtením tohoto offsetu, například pomocí instrukce `Add`.

Relace uspořádání „předek na nulovém offsetu" na třídách tvoří orientovaný les (nikoli obecný graf jako relace „předek" v C++) a je jí využíváno při hledání společného předka na instrukcích `PHINode` a dalších.

Výstupem analýzy je jednak paměťová reprezentace typů vzhledem k instrukcím, kterou využívají další analýzy, jednak textový výpis pro ruční kontrolu ve formě metadat k LLVM instrukcím.

6.4.6 Detekce volání virtuálních funkcí

Vhodnou demonstrací praktického využití předchozích analýz je detekce volání virtuálních funkcí. Ve všech zkoumaných překladačích byl vypozorován stejný vzor pro volání virtuálních funkcí C++. Nejprve je načtena ze třídy adresa virtuální tabulky, na ní adresa funkce, která je následně zavolána.

Pokud se pomocí předchozí analýzy podaří najít typ ukazatele na funkci v instrukci `Call`, lze tuto posloupnost instrukcí nahradit jinou posloupností tak, aby i po převedení LLVM IR do jazyka vyšší reprezentace v backendu bylo jasně patrné, že je volána virtuální funkce na určité pozici určité virtuální tabulky.

6.5 Výstup

Primárním výstupem je pozměněný LLVM IR, ze kterého je generován v backendu jazyk vyšší formy reprezentace. Kromě toho je generovaný tzv. soubor deskriptoru ve formátu JSON [8], který shrnuje všechny získané informace ze všech analýz C++. V deskriptoru jsou uvedeny následující výsledky analýz:

Seznam tříd (classes)

- Jméno
- Seznam jmen konstruktorů
- Seznam jmen destruktorů
- Seznam dvojic jméno nadřazené třídy – offset
- Seznam dvojic jméno tabulky virtuálních funkcí – offset

Seznam funkcí

- Jméno
- Adresový rozsah
- Seznam globálních proměnných, do kterých zapisuje
- Seznam globálních proměnných, ze kterých čte bez předchozího zápisu

Seznam tabulek virtuálních funkcí

- Generované jméno
- Adresový rozsah
- Seznam položek
  - Jméno funkce
  - Počáteční adresa funkce
  - Pokud je adaptér, jméno cílové funkce

Kapitola 7

Implementace

Kromě malé úpravy přední části jsou všechny analýzy implementovány za pomocí průchodů kódem LLVM IR ve střední části, využívajíce prověřené architektury nástroje `opt` z projektu LLVM.

Před začátkem psaní této práce sloužil middle end pouze pro optimalizaci kódu a byly v něm skoro výhradně průchody pocházející z projektu LLVM. V souvislosti s mojí i dalšími souběžně vznikajícími pracemi bylo rozhodnuto o posílení role střední části. Ta nově obsahuje i analýzy přímo pro dekompilátor, buď nově napsané, jako je případ této práce, nebo převedené z přední čí zadní části. Všechny nové průchody v middle endu jsou roztříděny do tří skupin: analýzy (analyses), optimalizace (optimalizations) a podpůrné průchody (utilities).

LLVM nabízí mocný framework pro vytváření i použití jednotlivých průchodů. Základním rozhraním jsou parametry příkazového řádku při volání programu `opt` – zde jsou zaregistrovány základní průchody, které se mají provést. Každý průchod může mít závisosti na dalších průchodech a výsledky jiných průchodů může zneplatňovat. Jeden průchod tak často bývá spouštěn opakovaně.

Průchody se liší podle rozsahu kódu, který mají možnost upravovat. V této práci jsou použity dva druhy: `ModulePass`, který má možnost upravovat veškerý kód modulu, a `FunctionPass`, který je spouštěn na jednotlivých funkcích. Existují však mnohé další, jak je popsáno ve [16].

Každý průchod je implementován C++ třídou, která dědí od abstraktního průchodu LLVM (v našem případě `ModulePass` nebo `FunctionPass`). Aby bylo možné ho používat, musí být zaregistrován pod unikátním argumentem pro příkazový řádek a popisným textem pro ladicí výpisy.

Tabulka 7.1 shrnuje všechny průchody, které jsem v rámci práce napsal, a obrázek 7.1 znázorňuje jejich vzájemné závislosti.

V dalším textu budou průchody popsány podrobněji.

7.1 Přední část

V přední části bylo nutné implementovat alespoň základní detekci tabulek virtuálních funkcí, tedy ukazatelů na funkce.

Proto byla vytvořena singletonová třída `VirtualTablesFinder`. Ta prochází datové sekce po jednotlivých adresách, a pokud hodnota na dané adrese odpovídá adresovému rozsahu kódové sekce, je dvojice adresa – hodnota uložena pro pozdější použití.

Průchod	Argument	Skupina	Předek
	Funkce		
Object File Loader	-objfileloader	utilities	ModulePass
	Zprostředkovává přístup k objektovému souboru pro další průchody		
Metadata Parser	-metadata	utilities	ModulePass
	Zpracovává vybraná metadata z frontendu		
VFTable Items	-vftableitems	analyses	ModulePass
	Vyhledává v datových sekcích spojité oblasti ukazatelů na funkce		
Constructor	-constructor	analyses	ModulePass
	Vyhledává konstruktory a destruktory a přiřazuje jim tabulky v.f.		
Ctor/Dtor	-ctordtor	analyses	ModulePass
	Analyzuje a třídí nalezené konstruktory/destruktory		
Hierarchy	-hierarchy	analyses	ModulePass
	Shlukuje spec. funkce a tabulky v.f. do tříd		
Global Access	-globalaccess	analyses	ModulePass
	Analyzuje přístup k globálním proměnným ve funkcích		
Adapter Methods	-adaptermethods	analyses	FunctionPass
	Detekuje adaptérové metody		
C++ Types	-cpptypes	analyses	FunctionPass
	Detekuje a propaguje nově získané typy		
Virtual Calls	-virtualcalls	analyses	FunctionPass
	Vyhledává a nahrazuje volání virtuálních metod		
Hierarchy Dump	-hierarchydump	utilities	ModulePass
	Ukládá všechny získané informace do souboru deskriptoru		

Tabulka 7.1: Průchody LLVM implementované v rámci této práce

K tomu dochází v upravené třídě `CallFunctionAnalysis`, kde jsou funkce označovány podle toho, zda jsou dosažitelné z funkce `main`. Všechny funkce, které začínají na adresách, které byly nalezeny v předchozím kroku, jsou automaticky označeny jako potencionálně dosažitelné. Tím se zabrání jejich možnému předčasnému odstranění.

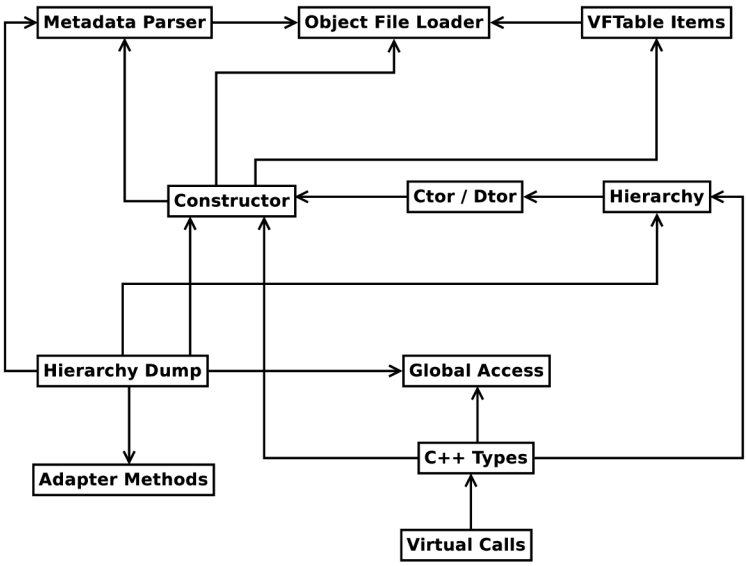
7.2 Object File Loader

Tento podpůrný průchod pouze zapouzdřuje načítání objektového souboru pomocí knihovny `FileFormatL`, sdílené s frontendem. Soubor je načten a uložen k dipozici následujícím průchodům.

7.3 Metadata Parser

LLVM IR poskytuje omezenou podporu pro anotaci částí kódu pomocí metadat. Tvoří je primitivní hodnoty, řetězce a jejich seznamy. Metadata mohou být pojmenovaná a mohou být přiřazena buď k modulu, nebo k instrukci. Neexistuje tedy možnost přímo je přiřadit například k funkci.

AVG decompiler využívá metadat k předávání doplňujících informací mezi jednotlivými stupni dekompilátoru. Metadata Parser soustřeďuje na jedno místo přístup k těmto metadatům, což je výhodné zejména vzhledem k možnému přejmenování klíčů.



Obrázek 7.1: Závislosti implementovaných průchodů. Šipky ukazují ke zdroji dat.

7.4 VFTable Items

Průchod provádí analýzu spojitých bloků ukazatelů na funkce v datových sekcích. První část algoritmu je v podstatě stejná, jako byla popsána v kapitole o přední části překladače, pouze s jinými datovými typy. Jsou opět prohledávány datové sekce na adresy ukazující do kódové sekce. Zde je navíc kontrolováno, zda na dané adrese začíná dekompilovaná funkce.

Druhá část pátrá po načítání adres virtuálních tabulek, jaké se vyskytují v konstruktorech a destruktorech. Akceptované fragmenty kódu jsou tyto:

```
; Zakladni:
store i32 <constant>, i32* %p_vftable

; Realne:
%addr = add i32 <constant>, 0
store i32 %addr, i32* %p_vftable

; Vysledek pruchodu Constructor:
store i32 ptrtoint (%vftable_x_type* @vftable_x to i32), i32* %p_vftable
```

Základní varianta ukazuje nejzákladnější uložení konstanty na místo v paměti. Konstanta bude označena jako potencionální začátek virtuální tabulky.

Bohužel tento průchod musí být schopen přijímat surový neoptimalizovaný kód z frontendu a ten většinou vypadá značně odlišně. Ten nejjednodušší z reálných případů ukazuje druhá varianta. Tento konkrétní příklad pochází z architektury x86, která umožňuje načtení celé adresy v jedné instrukci. Jiné jednodušší architektury na to potřebují procesorových instrukcí více, což po průchodu přední částí překladače může být reprezentováno až desítkami různých matematických i paměťových instrukcí v neoptimalizovaném LLVM IR. Proto byla vyvinuta třída `ConstantEvaluator`, která vyhodnocuje tyto stromy instrukcí, které ve výsledku vracejí konstantu.

Infrastruktura LLVM nezaručuje, že tento průchod bude spuštěn jen jednou – naopak, jakákoli následující analýza může výsledky průchodu zrušit, IR však zůstává změněný. Je proto nutné akceptovat i výsledky tohoto a navazujících průchodů, což ukazuje poslední příklad.

Průchod neprovádí sám o sobě žádné modifikace do LLVM IR.

7.5 Constructor

Průchod s názvem Constructor přímo navazuje na předchozí analýzu. Využívá seznam detekovaných instrukcí `Store` a souvislých ukazatelů na funkce, aby vytvořil v IR struktury reprezentující jednotlivé tabulky. Pro každou tabulku je vytvořen jeden unikátní typ a jedna globální instance, která obsahuje ukazatele na dekompilované funkce.

Instrukce `Store`, která původně ukládala konstantu, je nahrazena uložením ukazatele na nově vytvořenou globální strukturu, jak již bylo ukázáno v příkladu kódu v minulé podpapitole.

Kromě změněného kódu je výstupem informace o tom, které virtuální tabulky byly použity ve kterých funkcích, čehož využívají další analýzy.

7.6 Ctor/Dtor

Průchod pojmenovaný Ctor/Dtor analyzuje kód adeptů na konstruktory a destruktory. Jak popisuje kapitola 5.2.4, konstruktory a destruktory obsahují kód generovaný překladačem, podle kterého lze získat informace o třídní hierarchii.

Implementace následuje návrh z 6.4.2. Nejprve každou funkci analyzuje dvakrát pomocí stavového automatu, jako konstruktor i jako destruktor. U konstruktoru jsou analyzovány instrukce prvního základního bloku v pořadí vykonávání, u destruktoru instrukce posledního bloku v opačném směru. Tímto způsobem lze pro obě možnosti použít stejný stavový automat.

V prvním stavu jsou hledány inicializace ukazatelů na virtuální tabulky:

```
store i32 ptrtoint (%vftable_x_type* @vftable_x to i32), i32* %p_vftable
```

Ve druhém stavu jsou zajímavá volání nadřazených metod (volané funkce musejí být taktéž kandidáty na konstruktor či destruktor):

```
call i32 @ClassD2(%struct.0* %p_class) #0
```

Poté, co jsou nalezeny v obou průchodech zatímavé instrukce, je vybrána vhodnější varianta. U validního C++ kódu není možné, aby byly detekovány nadřazené metody v obou průchodech. Je-li tedy volání nadřazené metody nalezeno právě v jednom průchodu, je rozhodnuto. Je ale také možné, že třída dané funkce nemá žádné předky, a proto nadřazené funkce nevolá. V takovém případě rozhoduje počet nalezených tabulek. Pokud i ten je stejný a funkce má jen jeden zákadní blok, pak je zaevidována jako konstruktor i jako destruktor zároveň.

Kromě samotné kategorizace funkcí je zjišťováno také rozvržení tříd. Pro každý ukazatel na tabulku je zkoumán jeho offset ve třídě a stejně tak ukazatel na `this` při volání nadřazené funkce. Pro zjištění offsetu u tabulky je hledán následující vzor před instrukcí `Store`:

```
%addr_superclass = add i32 %addr_subclass, <offset>
%p_superclass = inttoptr i32 %addr_superclass to i32*
store i32 ptrtoint (%vftable_x_type* @vftable_x to i32), i32* %p_superclass
```

Podobný postup lze použít i při volání superkonstruktorů (nebo destruktorů):

```
%addr_superclass = add i32 %addr_subclass, <offset>
%p_superclass = inttoptr i32 %addr_superclass to i32*
call i32 @SuperclassConstructor(i32* %p_superclass) #0
```

Tento postup však nefunguje, pokud je použita volací konvence `__thiscall`. V takovém případě je třeba najít nejbližší předchozí zápis do `ECX` ve stejném základním bloku a na něj již aplikovat známý postup hledání instrukce `Add`.

Pokud při hledání kteréhokoli offsetu instrukci `Add` nenalezneme, považujeme příslušný offset za nulový.

Výstupem průchodu Ctor/Dtor je pro každou kandidátskou funkci struktura, která obsahuje:

- Zda se jedná o konstruktor, destruktor, nebo oba;
- Použité tabulky virtuálních funkcí a offsety jejich ukazatelů;
- Volané supermetody a offsety jejich tříd.

7.7 Hierarchy

Průchod Hierarchy neanalyzuje samotný LLVM IR, využívá však výsledky předchozích analýz, ze kterých vytváří kompletní obraz o třídách a jejich hierarchii v dekompilovaném programu.

Konstruktory a destruktory získané v průchodu Ctor/Dtor jsou seskupeny podle stejné množiny tabulek virtuálních funkcí a pro každou skupinu je vytvořena interní reprezentace detekované C++ třídy: `Hierarchy::Class`.

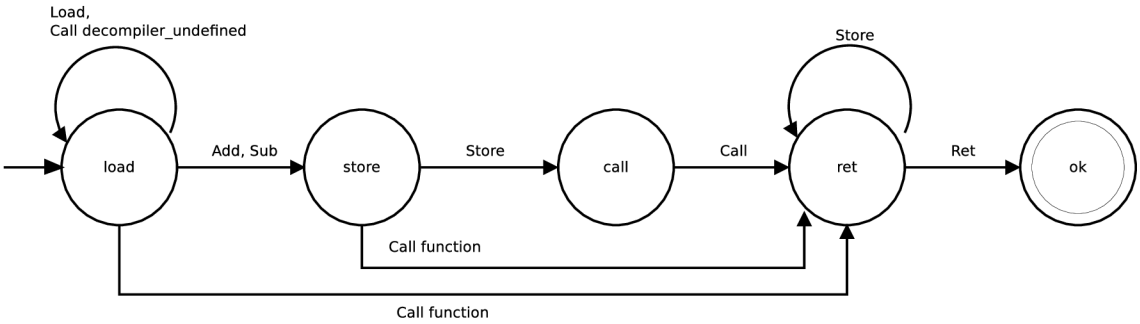
Pro každou detekovanou třídu je vytvořena struktura i v LLVM IR, která obsahuje ukazatele na použité virtuální tabulky. Tato struktura není programově nikde dále používána a z pohledu fyzického rozložení dat není přesná – ukazatele na tabulky nejsou na offsetech, na kterých byly detekovány. Jedná se ale o cennou informaci pro uživatele, čtenáře dekompilovaného kódu. Z kódu samotného může poznat, které třídy byly nalezeny a které tabulky k nim patří.

Každá detekovaná třída je pojmenovaná. Program se snaží využít názvy z tabulky symbolů či informací pro debugger, kde jsou uložena jména funkcí. Jméno třídy je tedy nejdelší společný začátek všech konstruktorů (a destruktorů bez znaku `~`) dané třídy. V případě, že informace o jménech k dispozici nejsou, je třída pojmenovaná podle adres svých virtuálních tabulek, například `Class_vftable_402204_vftable_402210`.

Výstupem analýzy je seznam tříd. Každá třída obsahuje tyto informace:

- Jméno;
- Seznam konstruktorů;
- Seznam destruktorů (může se překrývat s konstruktory, viz výše);
- Seznam virtuálních tabulek a jejich offsetů;
- Seznam nadtříd a jejich offsetů;
- Strukturu s virtuálními tabulkami.

Virtuální metody třídy lze zjistit z tabulek virtuálních funkcí. Nevirtuální metody nejsou detekovány.



Obrázek 7.2: Stavový automat použitý v analýze Adapter Methods

7.8 Global Access

Průchod Global Access zjišťuje, která funkce využívá které globální proměnné. Detekuje zápis do nich a jejich čtení bez předchozího zápisu. Obě analýzy operují nad funkcemi a fungují transitivně vzhledem k relaci volání funkce – jsou uvažovány i charakteristiky funkcí volaných ve zkoumané funkci. Tento průchod dává ostatním analýzám informaci o tom, zda volání nějaké funkce využívá hodnotu v globální proměnné a zda tato hodnota bude voláním funkce přepsána.

Průchod využívá výsledky analýzy `CallGraph`, již implementované v LLVM, která podává informace o vzájemném volání funkcí. Dále je využita funkcionalita spojitých komponent v grafu (Single Connected Component), která vyhledává v grafu cyklické závislosti a umožňuje ho procházet v efektivním pořadí tak, aby se co nejméně výpočtů muselo opakovat.

Implementace následuje již představený algoritmus 6.3 v kapitole Návrh. Analýza jedné funkce probíhá po základních blocích ve zpětném pořadí. Začíná se od koncových bloků směrem k předchůdcům. Cykly jsou ošetřeny tak, že do již zpracovaného bloku se vstupuje znovu pouze tehdy, došlo-li ke změně.

V rámci každého bloku se postupuje zpětně a do seznamů se ukládají globální proměnné, do kterých se zapisuje a ze kterých se čte (zápis maže čtení, protože to již není neinicializované).

Výstupem jsou pro každou funkci seznam write, obsahující zápisy, a seznam read, obsahující čtení neinicializované hodnoty.

7.9 Adapter Methods

Adaptérové metody mají v kódu generovaném všemi překladači podobnou strukturu, kterou lze zachytit stavovým automatem 7.2. Výstupem je pro každou funkci informace o tom, zda je adaptérem a jaká je jeho cílová funkce.

V prvním stavu jsou přeskakována volání funkcí generovaná dekompilátorem. Tyto funkce mají zvláštní význam a nemají oporu v původním strojovém kódu, pro tuto analýzu proto nejsou důležité.

Nejjednodušší adaptér obsahuje dále instrukce `Call` a `Ret`. Funkce s pouze těmito instrukcemi sice nemá žádný praktický význam, přesto ji můžeme považovat za platný adaptér. Typický adaptér obsahuje instrukce `Add/Sub` pro přetypování prvního parametru (ukazatele na `this`) a dále `Call` a `Ret`. Při konvenci `__thiscall` je před voláním trojice instrukcí

`Load`, `Add/Sub`, `Store`, protože ukazatel na `this` je uložen v registru. Poté následuje `Call` a původní nepřetypovanou hodnotu do registru vrací instrukce `Store`.

7.10 C++ Types

Průchod C++ Types provádí typování ukazatelů nalezených v LLVM IR. Tato analýza se zaměřuje výhradně na určení typů a neklade si za cíl jakkoli modifikovat typy hodnot v IR.

7.10.1 Struktura typu

Průchod vytváří pro svoji funkci typovou strukturu paralelní k typovému systému LLVM. Tato struktura je zamýšlena především pro interní potřebu propagace typů, avšak je přístupná jako výstup i dalším průchodům.

Struktura C++ typu obsahuje tyto položky:

- LLVM typ;
- Offset;
- Rodičovský C++ typ;
- Kořenovou instrukci, kde tento typ vznikl;
- Iteraci algoritmu, kdy typ vznikl;
- Poslední instrukci a hodnotu, odkud byl typ odvozen.

LLVM typ označuje bázový typ s možným offsetem, jedná-li se o ukazatel. Rodičovský typ označuje typ, z něhož byl aktuální typ odvozen (např. dereferencí). Kořenová instrukce označuje bod v programu, kde byl tento typ poprvé rozeznán, a spolu s iterací se používá k řešení konfliktů. Poslední instrukce a hodnota slouží pro ladicí výpisy.

Typy přiřazené k LLVM hodnotám se v průběhu vykonávání programu mění. Uvedená struktura je tedy vždy přiřazena nejen k LLVM hodnotě, ale také k instrukci, na které platí.

7.10.2 Získání typů

Implementované analýzy podporuje jediný zdroj typů, a sice konstruktory a destruktory. Různé překladače předávají ukazatel na alokovanou paměť konstruktorům různě (parametrem nebo v registru). Stávající implementace neodvozuje své chování od konkrétního typu překladače, namísto toho se pokouší využít všechny možnosti.

Je-li proto konstruktor či destruktor volán s alespoň jedním parametrem, je propagován typ tohoto parametru jako ukazatel na třídu (například `*Trida`). Navíc se v rámci stejného základního bloku vyhledává předchozí zápis do registru ECX, a pokud je nalezen, je globální proměnné přiřazen typ ukazatel na ukazatel na třídu (`**Trida`) a tento je také dále propagován.

7.10.3 Propagace typů v rámci základního bloku

Při volání konstruktorů často v praxi dochází k zrcadlení ukazatelů (pointer aliasing, dva nebo více rozdílných ukazatelů ukazují na stejné místo v paměti). Z tohoto důvodu je

třeba získané typy propagovat nikoli pouze dolů (po směru vykonávání programu), ale také nahoru, aby bylo správně otypované místo, kde se ukazatel poprvé zrcadlil.

Některé instrukce mají definovanou zvláštní sémantiku a omezení, která umožňují propagaci napříč jejich argumenty a návratovou hodnotou:

Instrukce přetypování LLVM IR je v našem případě je generovaný z assemleru, který typy proměnných nerozlišuje – k hodnotě v registru lze jednou přistupovat jako k číslu a v další instrukci jako k adrese do paměti. Aby byl IR validní, musí být vygenerována velká množství přetypování, která v původním assembleru nebyla a z hlediska analýza datového toku jsou nepodstatná. Proto je nový C++ typ beze změny propagován z operandu na návratovou hodnotu i naopak.

Instrukce `Load` Pro instrukci `Load` platí, že typ operandu musí být ukazatel na typ návratové hodnoty. Propagace z operandu na návratovou hodnotu tak obsahuje dereferenci. Naopak propagace na operand způsobuje referenci, a navíc dochází k propagaci typu přes paměť – jsou hledány předchozí i následující přístupy na stejné paměťové místo a ty jsou také případně otypovány. Zde jsou využity výsledky průchodu Global Access, který určuje, zda typ překročí přes instrukci `Call` – volání funkce.

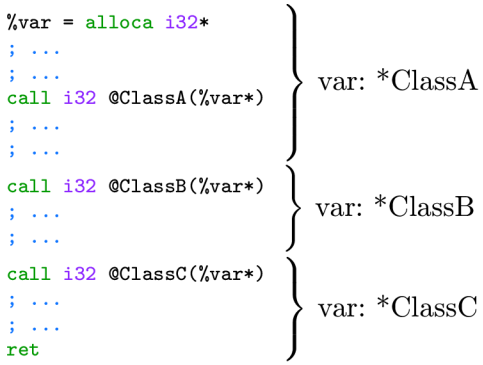
Instrukce `Store` Pro operandy instrukce `Store` platí podobná omezení, jako tomu bylo u instrukce `Load`. Typy jsou propagovány v obou směrech (ukazatel na hodnotu i naopak) a v obou případech dochází i k propagaci přes paměť směrem dolů.

Instrukce `Add` V kontextu typů C++ může sčítání znamenat přetypování na předka (upcast), nebo přístup k položce struktury (či třídy nebo tabulky virtuálních funkcí). Propagace probíhá jen od operandu k návratové hodnotě. Nejprve je zkoušeno přetypování: Pokud na konstantním offsetu aktuálního typu, daném operandem instrukce `Add`, nalezneme nadtřídu, je dále propagován typ této nadtřídy. V opačném případě je offset uložen k aktuálnímu typu.

Instrukce `GetElementPtr` Instrukce slouží pro získání ukazatele na element struktury. Má podobnou sémantiku jako `Add`, pouze offset není určen konstantou v bytech, nýbrž pořadím prvku ve struktuře. Pro účely této analýzy je nutno offset nejprve převést na byty, a pak se již postupuje stejně jako v případě `Add`.

Instrukce `PHINode` Phi instrukce jsou v LLVM využívány na začátku základních bloků pro výběh hodnoty podle bloku, ze kterého bylo předáno řízení. V kontextu typů platí omezení, že výsledný typ musí být společným předkem všech operandů v relaci „předek na nulovém offsetu," jak bylo popsáno v kapitole 6.4.5.

Instrukce `Select` `Select` slouží k jednoduché implementaci ternálního operátoru, provádí tedy výběr ze dvou variant na základě podmínky. Pro variantní operandy a návratovou hodnotu platí tedy stejná omezení jako pro operandy PHINode – návratový typ je nadtřídou obou variant.



Obrázek 7.3: Obory platnosti zdrojových instrukcí při propagaci typů

7.10.4 Dereference

K dereferenci dochází při propagaci typů na instrukcích `Load` a `Store`. Nejjednodušším případem samozřejmě je dereference ukazatele, výsledek je tím daný. V kontextu C++ je však nutné vyřešit případ, kdy výchozím typem je struktura nebo třída.

Struktura Je třeba vzít v úvahu offset, zjistit, jaký prvek se na daném offsetu struktury nachází, a pokud je to ukazatel, dereferencovat ten.

Třída Primárně kontrolujeme, zda offset neodpovídá offsetu některé z virtuálních tabulek, pokud ano, je výsledkem tato tabulka. V druhém kroku jsou kontrolovány nadtřídy stejným postupem. Pokud se nepodaří typ určit, je propagace zastavena.

7.10.5 Řešení konfliktů

Vzhledem k podstatě propagace typů dochází velmi často k pokusům o definici již existujícího typu na existující instrukci. Pokud se nový typ liší od stávajícího, je třeba rozhodnout, zda ho přepsat, či nikoli. K tomu slouží zdrojová instrukce každého typu.

Základní myšlenkou algoritmu je fakt, že každá zdrojová instrukce přednostně platí na instrukcích po ní následujících, ale obor platnosti může být omezen, pokud po ní následuje nová zdrojová instrukce. Pokud je zdrojová instrukce první v základním bloku, šíří se typy i před ni. Situaci ilustruje výpis 7.3. Pokud jsou zdrojové instrukce obou typů stejné, má přednost typ původní.

7.10.6 Propagace typů napříč bloky

Typy jsou propagovány napříč základními bloky pouze ve směru vykonávání programu. Na začátku zpracovávání každého bloku jsou nejprve evaluovány všechny hodnoty a jejich typy na koncových instrukcích všech předcházejících bloků a v případě konfliktů je pro danou množinu konfliktních typů vybrána jejich společná nadtřída, pokud taková existuje.

Protože základní bloky mohou obsahovat cykly, je někdy třeba provádět propagaci opakovaně. Pro tento účel má každý typ u sebe index iterace a typy z pozdější iterace vždy ruší ty předchozí.

7.10.7 Výstupy

Primárním výstupem analýzy je tabulka v paměti, která umožňuje následujícím analýzám zjistit typ dané hodnoty na dané instrukci. Především kvůli ladění jsou navíc detekované typy uloženy u dané instrukce v metadatech LLVM IR jako řetězce. Jiné změny LLVM IR nejsou prováděny.

7.11 Virtual Calls

Analýza volání virtuálních funkcí přímo navazuje na detekci typů. Vyhledává v kódu IR konstrukce, které odpovídají voláním virtuálních funkcí, a nahrazuje je čitelnějším zápisem. Posloupnost instrukcí, které jsou vyhledávány, odpovídá tomuto vzoru:

```
%p_class = inttoptr i32 %addr_class to i32*
%addr_vftable = load i32* %p_class, align 4

%p_vftable = inttoptr i32 %addr_vftable to i32*
%addr_function = load i32* %p_vftable, align 4
%p_function = inttoptr i32 %addr_function to i32 (i32)*
%result = call i32 p_function(i32 %addr_class) #0
```

Pokud tomu detekované typy odpovídají, jsou tyto instrukce nahrazeny instrukcemi podobnými následující šabloně:

```
%p_class = inttoptr i32 %addr_class to i32*
%addr_vftable = load i32* %p_class, align 4

%p_vftable = inttoptr i32 %addr_vftable to %vftable_8048a08_type*
%pp_function = getelementptr %vftable_8048a08_type* %p_vftable, i32 0, i32 0
%p_function = load void ()** %pp_function
%p_function_conv = bitcast void ()* %p_function to i32 (i32)*
%result = call i32 %p_function_conv(i32 %addr_class) #0
```

7.12 Hierarchy Dump

Tento průchod ukládá výsledky všech popsaných analýz do strojově čitelného souboru deskriptoru ve formátu JSON.

Kapitola 8

Experimenty

Správná činnost analýz byla průběžně testována pomocí sady testů se vzrůstající komplexností. Velká část z nich byla zahrnuta do tzv. regresních testů, tedy takových testů, které jsou určeny pro průběžnou kontrolu správné činnosti dekompilátoru během vývoje. V této kapitole popíši jednotlivé testovací scénáře a vyhodnotím jejich výsledky.

8.1 Regresní testy

Úkolem regresních testů je zajistit, aby nedocházelo k regresím – znovuzavedení již jednou opravené chyby či znefunkčnení již fungující analýzy při implementaci nových prvků. Do sady regresních testů tak mohou být zařazeny pouze ty testovací scénáře, které bezpodmínečně fungují. Jak bude ukázáno dále, v některých z mých testů existují konfigurace, které fungují pouze částečně, tyto proto nejsou do sady regresních testů zahrnuty, přesto je v této části okomentuji.

Testovány jsou tyto konfigurace – kombinace architektur a překladačů. S výjimkou Visual C++ jsou podporována platformy Windows (PE) a Linux (ELF):

- x86: GCC (ELF, PE), Clang (ELF, PE), Microsoft Visual C++ (PE);
- ARM: GCC (ELF, PE), Clang (ELF);
- MIPS: GCC (ELF), Clang (ELF);
- Thumb: GCC (ELF), Clang (ELF);
- PowerPC: GCC (ELF), Clang (ELF).

Součástí běhu testů je kompilace zdrojového kódu C++ danými překladači pro dané architektury a platformy, následná dekompilace a vyhodnocení výsledků. Výjimkou je Visual C++, pro který byl program ze stejných zdrojů skompilovaný předem, přiložen k testu jako vstup a test ho pouze dekompiluje. Z testů je vyloučena architektura Pic32, kterou dekompilátor AVG podporuje, ale v době psaní této práce je pro něj licencován pouze překladač C, nikoli C++.

Všechny testovací scénáře jsou nastaveny na úroveň optimalizací -O1. Je to proto, že všechny testovací vstupy jsou programově velmi jednoduché: většinou vypisují konstantní text. Při vyšší úrovni optimalizace překladač zoptimalizuje kód na několik výpisů na obrazovku ve funkci `main` a třídy či virtuální tabulky nejsou do strojového kódu vůbec přeloženy.

```
class ClassA {
public:
  int a, b;
  ClassA();
  virtual void doSomething();
};
ClassA::ClassA() : a(1), b(2) {
  printf("ClassA::ClassA\n");
}
void ClassA::doSomething() {
  printf("%i %i\n", a, b);
}
```

```
{
  "classes" : [{
    "constructors" : [ "ClassA" ],
    "destructors" : [ "ClassA" ],
    "name" : "ClassA",
    "superClasses" : [],
    "virtualFunctionTables" : [{
      "offset" : 0,
      "reference" : "vftable_87a0"
    }],
  "virtualFunctionTables" : [{
    "endAddr" : 34724,
    "items" : [{
      "adapterTo" : "",
      "name" : "ClassA__doSomething",
      "targetAddress" : 34492
    }],
    "name" : "vftable_87a0",
    "startAddr" : 34720
  }]
}
```

Obrázek 8.1: Jednoduchá třída. Vlevo zdrojový C++ kód, vpravo detekovaná hierarchie.

Většina testovacích scénářů se zaměřuje na správnou detekci hierarchie a nijak nezkoumá dekompilovaný kód jako takový. Výjimkou je poslední testovací scénář (virtual-calls), který zkoumá i kód.

V názvu každé z podkapitol je v závorce uvedeno jméno regresního testu.

8.1.1 Jednoduchá třída (simple)

Základní test s jedinou třídou s jedním konstruktorem, žádným destruktorem a jednou virtuální metodou. Zdrojový kód třídy a očekávaný výsledek zobrazuje výpis 8.1.

V získané hierarchii stojí za zmínku fakt, že nalezená speciální funkce je klasifikována jako konstruktor i destruktor. Toto chování je očekávané (viz 6.4.2).

Očekávaného výsledku dosahuje většina konfigurací překladačů a architektur. Problém je s kombinací PowerPC – Clang – ELF, ve které nejsou detekovány virtuální tabulky ani třídy. Pouze částečný úspěch zaznamenává architektura Thumb, kde selhává detekce ukazatelů na funkce ve frontendu, a virtuální funkce jsou tak vyoptimalizovány. Pokud je překlad spuštěn s parametrem `-k`, který ve výsledném kódu ponechává všechny funkce, je výsledek v pořádku.

8.1.2 Složitější třída (simple2)

Tento test vychází z prvního testu, avšak třída má konstruktor i destruktor a obě tyto funkce obsahují více základních bloků. Zdrojový kód třídy a očekávaný výsledek zobrazuje výpis 8.2.

Tento test vznikl pro ověření činnosti analýzy Ctor/Dtor při složitějších podmínkách. Vidíme, že funkce `ClassA` (původně konstruktor) a `ClassA_1` (destruktor) jsou detekovány jako konstruktor i jako destruktor, což je v souladu s očekáváním.

Tato analýza přijímá silně optimalizovaný LLVM IR kód, kde jsou architekturní rozdíly již neznatelné. Není proto překvapením, že výsledky jsou identické s výsledky předchozího testovacího scénáře.

```
class ClassA {
public:
  int a, b;
  ClassA();
  ~ClassA();
  virtual void doSomething();
};
ClassA::ClassA() : a(1), b(2) {
  int x;
  scanf("%i", &x);
  if (x == 0) printf("0");
  printf("ClassA::ClassA\n");
}
ClassA::~ClassA() {
  int x;
  scanf("%i", &x);
  if (x == 0) printf("0");
  printf("ClassA::~ClassA\n");
}
void ClassA::doSomething() {
  printf("%i %i\n", a, b);
}
```

```
{
  "classes" : [{
    "constructors" : [ "ClassA", "ClassA_1" ],
    "destructors" : [ "ClassA", "ClassA_1" ],
    "name" : "ClassA",
    "superClasses" : [],
    "virtualFunctionTables" : [{
      "offset" : 0,
      "reference" : "vftable_8048868"
    }]
  }],
  "virtualFunctionTables" : [{
    "endAddr" : 134514796,
    "items" : [{
      "adapterTo" : "",
      "name" : "ClassA__doSomething",
      "targetAddress" : 134514448
    }],
    "name" : "vftable_8048868",
    "startAddr" : 134514792
  }]
}
```

Obrázek 8.2: Třída s destruktorem. Vlevo zdrojový C++ kód, vpravo detekovaná hierarchie.

8.1.3 Vícenásobná dědičnost (multiple)

Test zahrnuje složitější třídní hierarchii – třídu dědící od dvou jiných tříd. Zdrojový kód a detekovaná hierarchie je zobrazena na výpisu 8.3.

Pro splnění tohoto testu musí být detekována nejen hierarchie samotná, ale také offsety nadřídí a tabulek virtuálních tabulek v třídě `ClassD3`. Speciální metoda `ClassD3` již musí být rozpoznána pouze jako konstruktor.

Ve výsledku překvapí kombinace ARM – GCC – PE, která pro nadřazené třídy generuje konstruktory dvakrát. Podobné chování bylo na Windows popsáno v [22], i když pro platformu x86. Protože detekovaná hierarhie odpovídá dekompilovanému kódu, je i tento výsledek považován za správný.

Testem prochází opět všechny konfigurace kromě již zmíněné PowerPC – Clang – ELF.

8.1.4 Adaptérové metody (multiple-adapters)

V tomto testu je ověřována správná analýza adaptérových metod, jak bylo popsáno v kapitole 7.9. Zdrojový C++ kód je až na detaily stejný, jako byl použitý v testu vícenásobné dědičnosti, a stejná je i očekávaná hierarchie. Výpisy již byly ukázány na 8.3.

Očekávaný výsledek se u tohoto testu liší pro Visual C++, který používá adaptérové metody v menší míře než GCC a Clang, a namísto nich provádí adaptaci přímo na začátku volaných metod (viz kapitola 5.1.6).

Testem neprochází opět konfigurace PowerPC – Clang – ELF, kde nejsou rozeznány třídy. U konfigurace ARM – Clang – ELF je rozeznána pouze jedna ze dvou adaptérových metod a u architektury Thumb žádná. V obou případech je důvodem fakt, že vygenerované adaptérové metody obsahují kromě očekávaných instrukcí ještě kód, se kterým stavový automat nepočítá. V případě Thumb se jedná o zbytečný zápis do zero flagu, který nemohl být vyoptimalizován, protože bezprostředně po něm následuje volání cílové funkce adaptéru. U platformy ARM po očekávaném kódu na konci adaptéru následuje konstrukce podobná

```
class ClassD1 {
public:
  virtual void method1();
  virtual void methodx();
  ClassD1();
  char c1;
};
class ClassD2 {
public:
  virtual void method2();
  virtual void methodx();
  virtual void methody();
  ClassD2();
  char c2;
};
class ClassD3: public ClassD1, public ClassD2{
public:
  virtual void method3();
  virtual void methodx();
  virtual void methody();
  ClassD3();
  char c3;
};
// implementace metod...
```

```
{
  "classes" : [{
    "name" : "ClassD1"
    // ...
  }, {
    "name" : "ClassD2"
    // ...
  }, {
    "constructors" : [ "ClassD3" ],
    "destructors" : [],
    "name" : "ClassD3",
    "superClasses" : [{
      "offset" : 0,
      "reference" : "ClassD1"
    }, {
      "offset" : 8,
      "reference" : "ClassD2"
    }],
    "virtualFunctionTables" : [{
      "offset" : 0,
      "reference" : "vftable_8048a08"
    }, {
      "offset" : 8,
      "reference" : "vftable_8048a20"
    }]
  }],
  "virtualFunctionTables" : [{
    "name" : "vftable_8048a20",
    "items" : [{
      "name" : "ClassD2__method2"
    }, {
      "adapterTo" : "ClassD3__methodx",
      "name" : "_ZThn8_N7ClassD37methodxEv"
    }, {
      "adapterTo" : "ClassD3__methody",
      "name" : "_ZThn8_N7ClassD37methodyEv"
    }]
  }, {
    "name" : "vftable_80489d8"
    // ...
  }, {
    "name" : "vftable_80489e8"
    // ...
  }, {
    "name" : "vftable_8048a08",
    "items" : [{
      "name" : "ClassD1__method1"
    }, {
      "name" : "ClassD3__methodx"
    }, {
      "name" : "ClassD3__method3"
    }, {
      "name" : "ClassD3__methody"
    }],
 }]
}
```

Obrázek 8.3: Složitější třída. Vlevo zdrojový C++ kód, vpravo detekovaná hierarchie (zkrácena)

volání virtuální funkce. Tato architektura by si jistě zasloužila hlubší analýzu, možná se jedná o ošetření výjimek.

8.1.5 Volání virtuálních funkcí (virtual-calls)

Zatímco všechny předchozí testy se zaměřovaly výhradně na detekovanou hierarchii, tento naopak zkoumá výsledný kód. Detailnímu popisu změn v dekompilovaném kódu bude věnována celá následující podkapitola.

V testu jsou využity informace z deskriptoru hierarchie o třídách a příslušných virtuálních tabulkách a ve výsledném kódu jsou hledána přetypování na struktury virtuálních tabulek, která jsou použita při dekompilaci virtuálních volání. Je vybrán příklad obsahující třídu s více virtuálními tabulkami a jsou vyzkoušeny všechny možnosti volání, včetně netriviálního přetypování na nadtřídu.

Tento test je bohužel velmi křehký, protože vyžaduje smysluplný LLVM IR z přední části, bezchybně detekovanou hierarchii (jak bylo testováno výše) a také musí fungovat správně analýza C++ Types a Virtual Calls. Úspěch tak mají pouze architektury x86 a MIPS. Kombinace x86 – GCC – PE je v době psaní této práce zatížena nesouvisející chybou v dereferenci struktur, která je již ve vstupu do analýz C++. Po opravení této chyby bude test velmi pravděpodobně procházet.

Kód generovaný pro Thumb a ARM obsahuje poměrně zvláštní konstrukce, jako jsou absolutní adresy (číselné konstanty) namísto ukazatelů na dynamicky alokovanou paměť. Pravděpodobně zde zapracovaly optimalizace překladačů pro funkci `main`. Selhává zde jak rozpoznání typů v přední části, tak následně i analýza C++ Types, jelikož typ nelze propagovat přes číselnou konstantu.

8.1.6 Shrnutí

Výsledky jednotlivých testů pro různé konfigurace shrnuje tabulka 8.1.

Arch.	Komp.	Platf.	simple	simple2	multiple	multiple adapters	virtual calls
x86	GCC	ELF	✓	✓	✓	✓	✓
		PE	✓	✓	✓	✓	(✓)
	Clang	ELF	✓	✓	✓	✓	✓
		PE	✓	✓	✓	✓	✓
	MSVC++	PE	✓	✓	✓	✓	✓
ARM	GCC	ELF	✓	✓	✓	✓	×
		PE	✓	✓	✓	✓	×
	Clang	ELF	✓	✓	✓	×	×
MIPS	GCC	ELF	✓	✓	✓	✓	✓
	Clang	ELF	✓	✓	✓	✓	✓
Thumb	GCC	ELF	✓*	✓*	✓*	×	×
	Clang	ELF	✓*	✓*	✓*	×	×
PowerPC	GCC	ELF	✓	✓	✓	✓	×
	Clang	ELF	×	×	×	×	×

* Vyžaduje přepínač -k

Tabulka 8.1: Výsledky testů

8.2 Srovnání původního a nového kódu

Srovnání provedu na kódech z testu Volání virtuálních funkcí. Vstupní kód obsahuje třídu dědící od dvou jiných tříd, celkem zde tedy existují 4 tabulky virtuálních funkcí (jedna na každou nadtřídu a dvě na poděděnou třídu). Na výpise A.1 je zdrojový kód v jazyce C++. Příklad ilustruje volání metod z různých tabulek. Je ukázáno i přetypování a volání funkce (`methodx`) přepsané ze dvou různých nadtříd.

Výpis A.2 ukazuje, jak tento kód dekompiluje AVG dekompilátor bez analýz popsaných v této práci. Pro příklad byla vybrána konfigurace x86 – Clang – ELF.

Poslední výpis A.3 ukazuje, jak je kód upraven pomocí nových analýz a transformací.

Na první pohled je vidět, že nový výpis obsahuje více funkcí. Veškeré C++ metody byly totiž dříve vyoptimalizovany během zpětného překladu jako nedostupné.

Nový kód na začátku obsahuje dvě sady struktur:

1. Struktury pro každou virtuální tabulku: uživatel studující zdrojový kód má ihned přehled o virtuálních tabulkách a jejich obsahu.

2. Struktury pro každou třídu: Objasňuje příslušnost tabulek ke třídám (i když bitové rozložení třídy je jiné)

V každém konstruktoru a destruktoru je původní nicneříkající číselná konstanta nahrazena ukazatelem na vygenerovanou strukturu virtuální tabulky.

Nejviditelnější změna v kódu je ve funkci `main`, kde jsou rozpoznána virtuální volání . Při troše studia je tak zřejmé, která virtuální metoda se nejspíše zavolá. Zde je již potřeba využít všechny dostupné informace obsažené v kódu i poznatky plynoucí z dědičnosti virtuálních tabulek. Nemusí být totiž volána vždy funkce té tabulky, která je dekódována u příslušného volání, ale i z jakékoli podřazené tabulky. Možnost volání různých implementací metody je ostatně důvodem celé existence pozdní vazby.

Pro pochopení výše uvedeného kódu je třeba poukázat na drobnou nepřesnost u dekódování operátoru `new`. Ten je rozložen na volání alokační funkce `_Znwj` a konstruktoru. Alokační funkce vrací ukazatel na alokovanou paměť jako svůj výsledek, avšak stávající verze dekompilátoru ho nerozpozná a namísto toho pracuje s výsledkem předchozího volání, v tomto případě funkce `puts`. Z toho plyne zdánlivě nelogické pojmenování `puts_rc` pro ukazatel na třídu.

Kapitola 9

Závěr

Cílem této práce byla analýza základních konstrukcí jazyka C++, především tříd a jejich hierarchie, virtuáních metod, konstruktorů a destruktorů. Tyto informace měly být získány pouze ze surového kódu, bez využití RTTI či dokonce přibalených informací pro debugger, které nemusejí být přítomny. Tohoto cíle bylo dosaženo.

V úvodních kapitolách byl popsán problém zpětného překladu a jako jeden z dekompilátorů byl zkoumán zpětný překladač AVG. Dále byly analyzovány základní konstrukce jazyka C++, byla prozkoumána jejich konkrétní implementace v různých překladačích, a to podle existujících článků i experimentálně. Na základě získaných informací byla navržena sada analýz a transformací ve formě průchodů kódem LLVM IR, které detekují vybrané konstrukce jazyka C++.

Mezi tyto konstrukce patří třídy, jejich konstruktory a destruktory a navázané tabulky virtuálních funkcí. Výstupem je popisný soubor, který je možné použít jako vstup pro další dekompilační nástroje, jako je například Hex Rays, pro nějž je právě týmem AVG dekompilátoru vyvíjen doplněk.

Pro demonstraci využití získaných informací byla vyvinuta analýza, která přiřazuje typy C++ tříd ukazatelům v dekompilovaném kódu a na jejich základě detekuje volání virtuálních funkcí. Přestože výsledný dekompilovaný kód má daleko k tomu, aby se vizuálně přiblížil jazyku C++, všechny potřebné informace pro analýzu například kódu viru jsou zde přítomny.

K dalšímu rozvoji se nabízí celá řada směrů. Pro získání co nejvěrnější dekompilace třídní hierarchie by jistě bylo vhodné zkombinovat zde popsané analýzy s informacemi z RTTI a informací pro debugger, jsou-li k dispozici.

V mé práci jsou detekovány základní konstrukce jazyka C++. V dalším rozvoji je možné pokračovat dále a dekompilovat další funktionality jazyka. Z těch bezprostředně logicky navazujících bych zmínil tyto:

- Čistě virtuální funkce (pure virtual functions);
- Virtuální dědičnost;
- Operátory `new` a `delete`.

Paralelně je možné pracovat na hlubší integraci mých a již dříve implementovaných analýz. Jako vhodný příklad se nabízí existující detekce parametrů pro volání funkcí. Ta dosud nemohla určit správně parametry u virtuálních funkcí, protože k nim neznala místa volání. Zde by bylo možné využít výstupy mojí analýzy Virtual Calls. Existující analýza složených typů by zase mohla využívat výsledky mojí analýzy C++ Types.

Ačkoli je moje práce hotová, projekt AVG dekompilátor je i nadále v aktivním vývoji. Proto jsem přesvědčen, že analýza C++, ke které jsem přispěl svým dílem, bude dále rozvíjena a některé mé návrhy budou uvedeny do praxe.

Literatura

[1] Adv. Programming Languages & Compilers – Data-Flow Analysis. [cit. 24. května 2015].
URL http://www.cs.columbia.edu/~aho/cs6998/lectures/12-09-11.pdf

[2] Boomerang decompiler. [cit. 24. května 2015].
URL http://boomerang.sourceforge.net

[3] Boomerang decompiler: What Boomerang can't do. [cit. 24. května 2015].
URL http://boomerang.sourceforge.net/cantdo.php

[4] C++ Language – C++ Tutorials. [cit. 24. května 2015].
URL http://www.cplusplus.com/doc/tutorial/

[5] IDA ClassInformer PlugIn. [cit. 24. května 2015].
URL http://sourceforge.net/projects/classinformer/

[6] Interactive DisAssembler. [cit. 24. května 2015].
URL https://www.hex-rays.com/products/ida/index.shtml

[7] Interactive DisAssembler. [cit. 24. května 2015].
URL https://www.hex-rays.com/products/decompiler/index.shtml

[8] JSON. [cit. 24. května 2015].
URL http://www.json.org/

[9] Lissom projekt. [cit. 24. května 2015].
URL http://www.fit.vutbr.cz/research/groups/lissom/

[10] The LLVM Compiler Infrastructure Project. [cit. 24. května 2015].
URL http://llvm.org/

[11] The LLVM Intermediate Representation. [cit. 24. května 2015].
URL http://llvm.org/docs/LangRef.html

[12] REC Studio 4 – Reverse Engineering Compiler. [cit. 24. května 2015].
URL http://www.backerstreet.com/rec/rec.htm

[13] SmartDec. [cit. 24. května 2015].
URL http://decompilation.info/

[14] __thiscall. [cit. 24. května 2015].
URL http://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx

[15] vtbl-ida-pro-plugin. [cit. 24. května 2015].
URL https://github.com/nektra/vtbl-ida-pro-plugin

[16] Writing an LLVM Pass. [cit. 24. května 2015].
URL http://llvm.org/docs/WritingAnLLVMPass.html

[17] Bruce, K. B.: *Foundations of Object-oriented Languages: Types and Semantics*. Cambridge, MA, USA: MIT Press, 2002, ISBN 0-262-02523-X.

[18] Cifuentes, C.; of Technology. School of Computing Science, Q. U.: *Reverse Compilation Techniques*. Queensland University of Technology, Brisbane, 1994.
URL http://books.google.cz/books?id=DWEFNQAACAAJ

[19] Eilam, E.: *Reversing: secrets of reverse engineering*. Wiley, 2005, ISBN 978-0764574818.

[20] Fokin, A.; Derevenetc, E.; Chernov, A.; aj.: SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, Washington, DC, USA: IEEE Computer Society, 2011, ISBN 978-0-7695-4582-0, s. 347–356, doi:10.1109/WCRE.2011.49.
URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6079860

[21] Fokin, A.; Troshina, K.; Chernov, A.: Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, Washington, DC, USA: IEEE Computer Society, 2010, ISBN 978-0-7695-4321-5, s. 240–243, doi:10.1109/CSMR.2010.43.

[22] Gray, J.: C++: Under the Hood. 1994.
URL http://www.openrce.org/articles/files/jangrayhood.pdf

[23] Hrbek, D.: *Strukturování kódu v zadní části zpětného překladače*. bakalářská práce, Brno, FIT VUT v Brně, 2014.

[24] ISO: International Standard ISO/IEC 14882:2011 – Programming Language C++. ISO ISO/IEC 14882:2011, International Organization for Standardization, Geneva, Switzerland, 2011.

[25] ISO: Working Draft, Standard for Programming Language C++. Iso, International Organization for Standardization, Geneva, Switzerland, 2014, [cit. 24. května 2015].
URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

[26] Křoustek, J.: *Analýza a převod kódu do vyššího programovacího jazyka*. diplomová práce, Brno, FIT VUT v Brně, 2009.

[27] Liberty, J.; Jones, B. L.: *Teach Yourself C++ in 21 Days (5th Edition)*. Indianapolis, IN, USA: Sams, 2004, ISBN 0672327112.

[28] Matula, P.: *Rekonstrukce datových typů při zpětném překladu kódu*. diplomová práce, Brno, FIT VUT v Brně, 2013.

[29] Mihulka, T.: *Zpětný překlad vybraných konstrukcí jazyka C++*. bakalářská práce, Brno, FIT VUT v Brně, 2014.

[30] Skochinsky, I.: Practical C++ Decompilation. 2011.
URL http://www.hexblog.com/wp-content/uploads/2011/08/Recon-2011-Skochinsky.pdf

[31] Stroustrup, B.: A History of C++: 1979 – 1991. *SIGPLAN Not.*, 1993: s. 271–297, ISSN 0362-1340, doi:10.1145/155360.155375.
URL http://www.stroustrup.com/hopl2.pdf

[32] Stroustrup, B.: Evolving a Language in and for the Real World: C++ 1991-2006. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-766-7, s. 4–1–4–59, doi:10.1145/1238844.1238848.
URL http://www.stroustrup.com/hopl-almost-final.pdf

Příloha A

Testovací příklady

A.1 Původní C++ kód

```
#include <stdio.h>


class ClassD1 {
public:
    virtual void method1();
    virtual void methodx();
    ClassD1();
    char c1;
};

class ClassD2 {
public:
    virtual void method2();
    virtual void methodx();
    virtual void methody();
    ClassD2();
    char c2;
};
class ClassD3 : public ClassD1, public ClassD2 {
public:
    virtual void method3();
    virtual void methodx();
    virtual void methody();
    ClassD3();
    char c3;
};

ClassD1::ClassD1() : c1('1') {
    printf("ClassD1::ClassD1\n");
}
ClassD2::ClassD2() : c2('2') {
    printf("ClassD2::ClassD2\n");
}
ClassD3::ClassD3() : c3('3') {
    printf("ClassD3::ClassD3\n");
}

void ClassD1::method1() {
    printf("ClassD1::method1\n");
}

void ClassD1::methodx() {
    printf("ClassD1::methodx c1=%c\n", c1);
}
```

```
void ClassD2::method2() {
    printf("ClassD2::method2\n");
}

void ClassD2::methodx() {
    printf("ClassD2::methodx c2=%c\n", c2);
}

void ClassD2::methody() {
    printf("ClassD2::methody c2=%c\n", c2);
}


void ClassD3::method3() {
    printf("ClassD3::method3\n");
}

void ClassD3::methodx() {
    printf("ClassD3::methodx c1=%c c2=%c c3=%c\n", c1, c2, c3);
}

void ClassD3::methody() {
    printf("ClassD3::methody c1=%c c2=%c c3=%c\n", c1, c2, c3);
}



int main() {
    printf("-*-* Test start *-*-\n");
    ClassD3 *d = new ClassD3();
    d->method1();
    d->method2();
    d->method3();

    ClassD1 *d1 = d;
    ClassD2 *d2 = d;
    d1->methodx();
    d2->methodx();

    d->methody();
    printf("-*-* Test end *-*-\n");

}
```

A.2 Dekompilovaný kód bez analýz C++

```
#include <stdint.h>
#include <stdio.h>

/* ------------- Structures ------------- */

struct struct_0 {
    int32_t e0;
    int8_t e1;
};

struct struct_2 {
    int32_t e0;
    int32_t e1;
    int8_t e2[1];
    int8_t e3;
};

/* -------- Function Prototypes --------- */

int32_t ClassD1(struct struct_0 * a1);
```

```
int32_t ClassD2(struct struct_0 * a1);
int32_t ClassD3(struct struct_2 * a1);

/* ------------- Functions -------------- */

// Address range: 0x8048640 - 0x8048668
int32_t ClassD1(struct struct_0 * a1) {
    // 0x8048640
    a1->e0 = 0x8048998;
    a1->e1 = 49;
    return puts("ClassD1::ClassD1");
}

// Address range: 0x8048670 - 0x8048698
int32_t ClassD2(struct struct_0 * a1) {
    // 0x8048670
    a1->e0 = 0x80489a8;
    a1->e1 = 50;
    return puts("ClassD2::ClassD2");
}

// Address range: 0x80486a0 - 0x80486dd
int32_t ClassD3(struct struct_2 * a1) {
    int32_t v1 = (int32_t)a1;
    ClassD1((struct struct_0 *)a1);
    ClassD2((struct struct_0 *)(v1 + 8));
    *(int32_t *)v1 = 0x80489c8;
    *(int32_t *)(v1 + 8) = 0x80489e0;
    a1->e3 = 51;
    return puts("ClassD3::ClassD3");
}

// Address range: 0x8048840 - 0x80488ae
int main(int argc, char ** argv) {
    int32_t puts_rc = puts("-*-* Test start *-*-"); // 0x804884a
    _Znwj();
    ClassD3((struct struct_2 *)puts_rc);
    ((int32_t (*)(int32_t))*(int32_t *)*(int32_t *)puts_rc)(puts_rc);
    int32_t v1 = puts_rc + 8; // 0x804886c
    ((int32_t (*)(int32_t))*(int32_t *)*(int32_t *)v1)(v1);
    int32_t v2 = *(int32_t *)(*(int32_t *)puts_rc + 8); // 0x804887c
    ((int32_t (*)(int32_t))v2)(puts_rc);
    int32_t v3 = *(int32_t *)(*(int32_t *)puts_rc + 4); // 0x8048884
    ((int32_t (*)(int32_t))v3)(puts_rc);
    ((int32_t (*)(int32_t))*(int32_t *)(*(int32_t *)(puts_rc + 8) + 4))(v1);
    int32_t v4 = *(int32_t *)(*(int32_t *)puts_rc + 12); // 0x8048895
    ((int32_t (*)(int32_t))v4)(puts_rc);
    puts("-*-* Test end *-*-");
    return 0;
}
```

A.3 Dekompilovaný kód s analýzami C++

```
struct ClassD1 {
    struct vftable_8048998_type * e0;
};

struct ClassD2 {
    struct vftable_80489a8_type * e0;
};

struct ClassD3 {
    struct vftable_80489c8_type * e0;
    struct vftable_80489e0_type * e1;
};

struct struct_0 {
```

```
    int32_t e0;
    int8_t e1;
};

struct struct_2 {
    int32_t e0;
    int32_t e1;
    int8_t e2[1];
    int8_t e3;
};

struct vftable_8048998_type {
    void (*e0)();
    void (*e1)();
};

struct vftable_80489a8_type {
    void (*e0)();
    void (*e1)();
    void (*e2)();
};

struct vftable_80489c8_type {
    void (*e0)();
    void (*e1)();
    void (*e2)();
    void (*e3)();
};

struct vftable_80489e0_type {
    void (*e0)();
    void (*e1)();
    void (*e2)();
};

/* -------- Function Prototypes --------- */

int32_t ClassD1(struct struct_0 * a1);
int32_t ClassD2(struct struct_0 * a1);
int32_t ClassD3(struct struct_2 * a1);
void ClassD1__method1(void);
void ClassD1__methodx(void);
void ClassD2__method2(void);
void ClassD2__methodx(void);
void ClassD2__methody(void);
void ClassD3__method3(void);
void ClassD3__methodx(void);
void _ZThn8_N7ClassD37methodxEv(void);
void ClassD3__methody(void);
void _ZThn8_N7ClassD37methodyEv(void);

/* ---------- Global Variables ---------- */

struct ClassD1 g1; // ClassD1_instance
struct ClassD2 g2; // ClassD2_instance
struct ClassD3 g3; // ClassD3_instance
struct vftable_8048998_type vftable_8048998 = {
    .e0 = ClassD1__method1,
    .e1 = ClassD1__methodx
}; // 0x8048998
struct vftable_80489a8_type vftable_80489a8 = {
    .e0 = ClassD2__method2,
    .e1 = ClassD2__methodx,
    .e2 = ClassD2__methody
}; // 0x80489a8
struct vftable_80489c8_type vftable_80489c8 = {
    .e0 = ClassD1__method1,
    .e1 = ClassD3__methodx,
```

```
    .e2 = ClassD3__method3,
    .e3 = ClassD3__methody
}; // 0x80489c8
struct vftable_80489e0_type vftable_80489e0 = {
    .e0 = ClassD2__method2,
    .e1 = _ZThn8_N7ClassD37methodxEv,
    .e2 = _ZThn8_N7ClassD37methodyEv
}; // 0x80489e0

/* ------------- Functions -------------- */

// Address range: 0x8048640 - 0x8048668
int32_t ClassD1(struct struct_0 * a1) {
    // 0x8048640
    a1->e0 = (int32_t)&vftable_8048998;
    a1->e1 = 49;
    return puts("ClassD1::ClassD1");
}

// Address range: 0x8048670 - 0x8048698
int32_t ClassD2(struct struct_0 * a1) {
    // 0x8048670
    a1->e0 = (int32_t)&vftable_80489a8;
    a1->e1 = 50;
    return puts("ClassD2::ClassD2");
}

// Address range: 0x80486a0 - 0x80486dd
int32_t ClassD3(struct struct_2 * a1) {
    int32_t v1 = (int32_t)a1;
    ClassD1((struct struct_0 *)a1);
    ClassD2((struct struct_0 *)(v1 + 8));
    *(int32_t *)v1 = (int32_t)&vftable_80489c8;
    *(int32_t *)(v1 + 8) = (int32_t)&vftable_80489e0;
    a1->e3 = 51;
    return puts("ClassD3::ClassD3");
}

// Address range: 0x80486e0 - 0x80486f8
void ClassD1__method1(void) {
    // 0x80486e0
    puts("ClassD1::method1");
}

// Address range: 0x8048700 - 0x804871f
void ClassD1__methodx(void) {
    // 0x8048700
    int32_t v1;
    printf("ClassD1::methodx c1=%c\n", *(int8_t *)(v1 + 4));
}

// Address range: 0x8048720 - 0x8048738
void ClassD2__method2(void) {
    // 0x8048720
    puts("ClassD2::method2");
}

// Address range: 0x8048740 - 0x804875f
void ClassD2__methodx(void) {
    // 0x8048740
    int32_t v1;
    printf("ClassD2::methodx c2=%c\n", *(int8_t *)(v1 + 4));
}

// Address range: 0x8048760 - 0x804877f
void ClassD2__methody(void) {
    // 0x8048760
    int32_t v1;
```

```
    printf("ClassD2::methody c2=%c\n", *(int8_t *)(v1 + 4));
}

// Address range: 0x8048780 - 0x8048798
void ClassD3__method3(void) {
    // 0x8048780
    puts("ClassD3::method3");
}

// Address range: 0x80487a0 - 0x80487cf
void ClassD3__methodx(void) {
    // 0x80487a0
    int32_t v1;
    int8_t v2 = *(int8_t *)(v1 + 12); // 0x80487ab
    int8_t v3 = *(int8_t *)(v1 + 13); // 0x80487af
    printf("ClassD3::methodx c1=%c c2=%c c3=%c\n", *(int8_t *)(v1 + 4), v2, v3);
}

// Address range: 0x80487d0 - 0x80487e8
void _ZThn8_N7ClassD37methodxEv(void) {
    // 0x80487d0
    ClassD3__methodx();
}

// Address range: 0x80487f0 - 0x804881f
void ClassD3__methody(void) {
    // 0x80487f0
    int32_t v1;
    int8_t v2 = *(int8_t *)(v1 + 12); // 0x80487fb
    int8_t v3 = *(int8_t *)(v1 + 13); // 0x80487ff
    printf("ClassD3::methody c1=%c c2=%c c3=%c\n", *(int8_t *)(v1 + 4), v2, v3);
}

// Address range: 0x8048820 - 0x8048838
void _ZThn8_N7ClassD37methodyEv(void) {
    // 0x8048820
    ClassD3__methody();
}

// Address range: 0x8048840 - 0x80488ae
int main(int argc, char ** argv) {
    int32_t puts_rc = puts("-*-* Test start *-*-"); // 0x804884a
    _Znwj();
    ClassD3((struct struct_2 *)puts_rc);
    ((int32_t (*)(int32_t))((struct vftable_80489c8_type *)*(int32_t *)puts_rc)->e0)(puts_rc);
    int32_t v1 = puts_rc + 8; // 0x804886c
    ((int32_t (*)(int32_t))((struct vftable_80489a8_type *)*(int32_t *)v1)->e0)(v1);
    ((int32_t (*)(int32_t))((struct vftable_80489c8_type *)*(int32_t *)puts_rc)->e2)(puts_rc);
    ((int32_t (*)(int32_t))((struct vftable_80489c8_type *)*(int32_t *)puts_rc)->e1)(puts_rc);
    ((int32_t (*)(int32_t))((struct vftable_80489a8_type *)*(int32_t *)(puts_rc + 8))->e1)(v1);
    ((int32_t (*)(int32_t))((struct vftable_80489c8_type *)*(int32_t *)puts_rc)->e3)(puts_rc);
    puts("-*-* Test end *-*-");
    return 0;
}
```

Příloha B

Obsah DVD

Přiložené DVD obsahuje:

- Elektronickou verzi tohoto dokumentu;
- Zdrojové kódy mojí práce;
- Předkompilované objektové soubory dekompilátoru;
- Testovací vstupy:
  - Zdrojové kódy;
  - Binární soubory pro různé platformy a architektury přeložené různými překladači;
  - Referenční výstupy.
- Makefile, kterým je možno přeložit soubory mojí práce, slinkovat dekompilátor a spustit dekompilaci testovacích vstupů.

Podrobný popis adresářové struktury je v souboru README.txt v kořenovém adresáři DVD.